A Program for Generating and Analyzing Term Rewriting Systems

by

Randy Forgaard
S.B., Massachusetts Institute of Technology
(1981)

Submitted to the Department of Electrical Engineering and Computer Science in partial fulfillment of the requirements for the degree of

Master of Science in Computer Science

at the

Massachusetts Institute of Technology

September 1984

Key words: REVE, Term Rewriting Systems, Knuth-Bendix Completion Procedure, Automatic Theorem Proving, Termination Orderings, Equational Theories, Inductive Theories, Inductionless Induction, Recursive Path Ordering, Recursive Decomposition Ordering, Extensible Path Ordering, Rewrite Rule Laboratory

A Program for Generating and Analyzing Term Rewriting Systems

by

Randy Forgaard

Submitted to the Department of Electrical Engineering
and Computer Science on 4 September 1984, In partial
fulfillment of the requirements for the Degree of
Master of Science in Computer Science

Abstract

This thesis presents new results in the use of term rewriting systems for automatic theorem proving. The design and implementation of REVE 2, a computer program that incorporates these results, is described. In addition, an introduction to the basic theory, procedures, and algorithms of term rewriting is provided, in a manner suitable for non-specialists.

A principal application of rewriting systems is reasoning about the equational and inductive theories associated with a finite set of axioms. In this context, the Knuth-Bendix completion procedure is typically used in the hope of constructing a confluent and terminating rewriting system from the axioms. Knuth-Bendix incrementally ensures termination by using a reduction ordering on terms to order equations into rewrite rules during the completion process. Serious impediments to the use of Knuth-Bendix in automatic proofs of equational and inductive theorems have been: 1) the need for user interaction, and 2) the lack of available state-of-the-art implementations.

REVE 2 reduces the need for user interaction in two ways. First, it uses *automatic orderings*, whose implementations automatically compute all of the possible valid extensions to the ordering that allow an unorderable equation to be ordered. Second, it uses a robust, task-based, *failure-resistant* Knuth-Bendix design that incorporates a fine-grained scheme for automatic equation postponement.

From the beginning, it has been a fundamental design goal to make REVE 2 a well-documented, highly-modular, easily-modified program, based on sound principles of software engineering. The user interface to REVE 2 has been designed for ease of use by both novice and expert.

Thesis Supervisor: Prof. John V. Guttag
Title: Associate Professor of Computer Science

Acknowledgments

I would like to express my foremost thanks to my advisor, John Guttag, for his unceasing encouragement and support throughout this thesis and my graduate career, for having sufficient faith in this fledgling research area to suggest this thesis topic to me, for his numerous technical insights, and for his uncanny ability to hone and reorganize my prose. I owe special thanks to Dave Detlefs, friend, officemate, visionary, realist, colleague, devil's advocate, patient audience, technical collaborator, tireless implementor, and Renaissance man, for taking over REVE 2, and for being crazy enough to suggest and implement automatic orderings.

Many people in the term rewriting community generously gave of their time to discuss ideas and share their knowledge with me, without which this work would not have been possible. I am especially grateful to Pierre Lescanne for REVE 1, for his encouragement and ideas for REVE 2, for the countless discussions, for carefully reading and commenting on this thesis, and for his great patience and responsiveness to my never-ending questions. The exchange of ideas with people at General Electric Corporate Research and Development, especially Deepak Kapur, has been particularly helpful. I want to thank David Plaisted for suggesting the closure ordering, and Lescanne and Guttag for bringing Plaisted's ordering to my attention. Many thanks to Jean-Pierre Jouannaud for answering my ordering questions. Those who attended the NSF Workshop on the Rewrite Rule Laboratory have provided valuable feedback and suggestions.

I am greatly indebted to everyone "on the hall" for the pleasant, supportive, and stimulating working environment. Kathy Yelick lent a sympathetic ear and brought fresh perspectives to many technical problems. Gene Stark, Ron Kownacki, Joe Zachary, and Jim Restivo gave me their good humor and technical expertise on many occasions. Bob Scheifler rescued my computer files when all hope was lost. Julie Lancaster offered her invaluable wisdom, friendship, and support through the good times and the bad.

Finally, I am very thankful to my parents, who have given me an optimistic outlook, a taste for quality, and total support for whatever I want to do.

Preface

This thesis documents the theory and design behind the REVE 2 term rewriting system generator. Though most of the new material contained in this document originated with the author, REVE is in no way a single-handed effort. It is a team project, reflecting the work of researchers from several laboratories and serving a growing international community of users.

An implementation of the Knuth-Bendix completion procedure was produced by John Goree [Goree 81] in John Guttag's Systematic Program Development (SPD) group at the MIT Laboratory for Computer Science. Though only a bare-bones implementation, it featured a modular design, and the subsidiary abstractions used by Knuth-Bendix were organized as layered "virtual machine" primitives.

REVE 1 [Lescanne 83a] was conceived and implemented by Pierre Lescanne[1], a researcher at the Centre de Recherche en Informatique de Nancy (CRIN) in France, during his visit with SPD in 1980-82. It included one of the first implementations of Knuth-Bendix to deal effectively and flexibly with rewriting system termination, making use of a new, incremental class of simplification orderings [Jouannaud 82a]. REVE 1 pioneered the idea of extending the ordering, as needed, during termination proofs. The program had an interactive interface that allowed users to enter equations and rewrite rules in conventional notation (including infix), and provided a number of user commands that allowed access to some basic rewriting and unification primitives. REVE 1 introduced important notions regarding the style and scope appropriate to a system for experimenting with term rewriting.

REVE 2 has been written from the ground up, using and expanding on the ideas in REVE 1. However, REVE 2 has the additional goals of providing 1) a solid source code base upon which to build, 2) automatic theorem proving capabilities, suitable for embedding in other applications, and 3) a friendly and powerful user interface. REVE 2 has been carefully modularized and documented to meet the first goal, including a complete set of data and procedural abstraction implementations that are pertinent to rewriting applications. We have made substantial progress toward the second goal by incorporating features into REVE 2 that

[1]The name "REVE," pronounced "rev," was chosen by Lescanne. *Rêve* is a French word, meaning "dream."

allow termination proofs and Knuth-Bendix to proceed nearly automatically. The third goal has been addressed with a flexible command interpreter that provides a rich set of commands, on-line help facilities, and detailed error messages. REVE is an "open system": anyone may obtain the source code and tailor it to their purposes. It is hoped that REVE 2 can serve as groundwork for implementation efforts by many researchers, permitting easier transference of algorithms among colleagues and expanded opportunities for experimentation.

The author designed and implemented the core of REVE 2, including the failure-resistant Knuth-Bendix, during 1982-83. David Detlefs, also of SPD, designed and implemented the EPOS automatic ordering, and has taken over primary responsibility for maintaining REVE. We have also greatly profitted from related theoretical and implementation work of colleagues in SPD, at CRIN, at General Electric Corporate Research and Development, at the State University of New York at Stony Brook, and at the University of Illinois at Urbana-Champaign.

REVE 2 is currently in use in many university and industrial laboratories in the United States and abroad. The source code and executable version of REVE, together with the CLU [Liskov 81] programming language in which it is implemented, are available for research and educational uses for a nominal distribution charge. REVE and CLU currently run on VAX[2] computers under Berkeley UNIX[3]. Inquiries should be sent to John V. Guttag, MIT Laboratory for Computer Science, 545 Technology Square, Cambridge, MA 02139.

In this thesis, we will refer to REVE 2 as simply REVE.

Randy Forgaard
September 1984

[2]VAX is a Trademark of Digital Equipment Corporation.

[3]UNIX is a Trademark of AT&T Bell Laboratories.

Table of Contents

Table of Figures

Chapter One

Introduction

1.1 Background

In recent years there has been a surge of interest in term rewriting systems. This has been sparked both by significant progress in understanding the theoretical aspects of rewriting systems and by the development of important new applications for these systems. These applications include automated deduction, program verification, specification analysis, program transformation, synthesis of programs, compilers, data base management systems, computer algebra systems, and the study of word problems in algebra, where term rewriting methods were first applied.

Term rewriting systems are often used to reason about the equational and inductive theories associated with a finite set of equations, called axioms. For nearly any interesting equational or inductive theory, the equivalence classes with respect to that theory are infinite. Proving that a particular equation is in the equational or inductive theory of a set of axioms is typically an ad-hoc process, using those axioms and the proof rules of equational and inductive reasoning. However, in some cases, a rewriting system with certain properties can be constructed for those axioms, enabling proofs to be effectively automated.

A rewriting system is a set of rewrite rules. Each rule is a "one-way" equation: if a term, or one of its subterms, matches the form of the left-hand side of the rule, the term or subterm may be "rewritten" to have the form of the rule's right-hand side. Every rewrite rule in a rewriting system for a set of axioms is in the equational theory of those axioms, so using a rule to rewrite a term is a valid inference in that equational theory. Once a term has been rewritten, one may further rewrite its rewritten form, to produce more rewritten forms, all of which are equivalent to the original term in the equational theory. A "normal form" for a term is a rewritten form of that term that cannot be rewritten further using any rule in the rewriting system. If all terms have a normal form with respect to the rewriting system, the rewriting system is said to "terminate." The rewriting system is "confluent" if, for any term, the normal form of that term is the same no matter what order the rules are applied, whenever a normal form exists. Rewriting systems that are both terminating and confluent are said to be "convergent."

To automate equational theorem proving, we are interested in finding a convergent rewriting system for a set of axioms. For such a rewriting system, an equation is in the equational theory of the axioms if and only if the normal forms of its two sides are the same. Unfortunately, both termination and confluence are undecidable, which complicates the problem of finding a convergent rewriting system for a given set of axioms. However, widely-applicable and easily-automated sufficient conditions for these two properties are known. A popular method for proving termination is to exhibit a "reduction ordering" on terms such that, for each rule in the rewriting system, the left-hand side is greater than the right-hand side under that ordering. Several such reduction orderings have emerged in recent years. Once termination has been established, confluence is decidable. When a terminating rewriting system is not confluent, one may use a special technique, called the Knuth-Bendix completion procedure, for adding additional rules to the system in the hope of achieving confluence. All rules added in this manner are in the equational theory of the original axioms, so the theorem proving utility of the rewriting system is preserved. When a convergent rewriting system for a set of axioms can be constructed in this manner, one has an efficient decision procedure for the equational theory of those axioms.

Convergent rewriting systems are also useful in automatically proving inductive theorems. To prove, by hand, that an equation is in the inductive theory of a set of axioms, one must inductively show that the theorem holds for all ground terms contructed from operators that appear in those axioms. However, if all such operators are completely defined with respect to the axioms, an "inductionless induction" approach may be used to prove inductive theorems. This automatic method consists of using Knuth-Bendix to construct a convergent rewriting system for the axioms together with the proposed inductive theorem. If such a rewriting system can be built, the proposed equation is an inductive theorem of the original axioms if and only if Knuth-Bendix finds no inconsistencies in the equational theory.

Of interest, then, is the availability of powerful and easy to use programs that incorporate implementations of reduction orderings and Knuth-Bendix. Some current systems that provide some of these capabilities are Affirm [Musser 80a], FORMEL [Huet 80a, Huet 82], RRL

[Kapur 84a], and [Göbel 84]. REVE, the subject of this thesis, differs from these programs by providing implementations of the pertinent procedures that allow theorem proving to proceed almost totally automatically.

1.2 Motivation for Building REVE

While the progress of research into rewriting systems has been significant, it has been impeded by the inordinate difficulty of implementing and using the increasingly complex procedures and algorithms prevalent in current term rewriting research. The crux of the problem is twofold: the large effort required to build state-of-the-art software, and the difficulty of acquiring usable software from others. The difficulty of acquiring or constructing good rewriting software serves both to slow down the work of those already involved in studying or using term rewriting systems and to inhibit the entry of new researchers into the field. It affects theoretical work as well as application-oriented work.

1.2.1 Building Applications

It is becoming increasingly likely that mechanical inference techniques based on term rewriting can be useful in a wide variety of applications. Unfortunately, it is exceedingly difficult for anyone who is not well versed in the theory of rewrite rule systems to make good use of them. Not only must one contend with all the normal problems that arise in relatively large software projects, but one is also faced with a number of problems peculiar to this kind of effort. Simply to program efficient implementations of the basic primitives requires:

(1) Conducting a literature search to find appropriate algorithms,

(2) Reading and understanding several papers that are almost certainly aimed at a relatively theoretically-minded audience,

(3) Choosing a representation for the primitive data objects and mapping the algorithms presented in papers (each of which is likely to have used different representations) onto those representations, and finally

(4) Implementing it all in some programming language.

After the primitives are implemented, the problem of understanding and implementing a growing number of useful but complex procedures, e.g., the Knuth-Bendix completion procedure or associative-commutative unification, remains. Once this rather lengthy digression is complete, one can finally begin working on application-related problems.

A major problem in acquiring software upon which one can build is that there is relatively little exportable software available. What there is, has, in general, been built for a particular use: to test a particular algorithm or to provide a particular facility. These programs rarely come with the hooks necessary to make them good building blocks. Pulling them together into a coherent system is almost impossible. They are written in different languages (mostly dialects of LISP), they use different representations of basic objects (e.g., terms), and they are often sparsely documented.

1.2.2 Theoretical Work

While accessing and understanding the relevant literature presents less of a problem to the theoretically-oriented than to those interested primarily in applications, the difficulties common to the production of all software are likely to present more of a problem. Certainly, the investment of considerable amounts of time in software development represents a serious digression for the theoretical group. Unfortunately, there are at least two excellent reasons why such a digression may seem useful or even necessary.

First, the manipulation of examples plays a vital role in much of the theoretical work in the rewrite rule area. Before trying to prove a difficult conjecture one often spends some time looking for a counterexample. If one finds such a counterexample, it may indicate a useful way to "patch" the conjecture. At the very least, it spares one the trouble of trying to prove a false conjecture. If one doesn't find a counterexample, an examination of why the examples tried were not counterexamples is often very helpful in constructing a proof of the validity of the conjecture. In a similar vein, one often develops new conjectures through the study of examples. It is sometimes possible to work these examples by hand, but doing so is generally too difficult to consider. The alternative of writing a program to experiment with an unproven idea is also usually seen as being prohibitively time-consuming.

The second reason is that it is difficult to judge the utility of much of the work in this area. Decision procedures don't exist for deciding most of the important questions about a rewriting system; e.g., is it terminating, is it confluent, is this or that theorem in its theory, etc.

Consequently, a great deal of effort has been devoted to the development of restricted classes of rewriting systems for which some questions are decidable, and to the development of semi-decision procedures for unrestricted sets of rewrite rules. The utility of such work often hinges on whether it deals with a significant subset of those sets of rules that arise in various applications. Even when a technique is in principle applicable to a wide class of rewriting systems, efficiency issues often arise. The worst case running time of many important procedures and algorithms is clearly prohibitive. This leads one to consider average case behavior. However, meaningful analytic results in this area can be exceedingly difficult to derive. One may have to consider such things as the number of rules, the size of the rules, the structure of the rules, etc. In many cases, a procedure's primary use is as a subroutine of some other procedure, and its efficiency is most productively studied in a specialized context established by the calling procedure.

The difficulty of judging the utility of new procedures and algorithms leads one to attempt empirical evaluation. Unfortunately, it is usually impossible to conduct useful experiments by hand. One has the choice of either implementing one's techniques and trying them on an appropriate data base of examples (which one will probably have to create), or merely speculating on the applicability of those techniques. Researchers in the field, confronted with the difficulty of doing the former, have almost invariably chosen the latter.

REVE has been designed to help meet the above needs of both theoreticians and potential users of rewriting applications. We hope it can facilitate the conducting of experiments with rewriting systems, supply the primitives needed for automatic theorem proving, and provide a firm base upon which to build application programs.

1.3 Overview of Thesis

This thesis introduces the basic theory and procedures related to term rewriting, presents new results in automatic theorem proving using rewriting systems, and describes the design and implementation of REVE, which incorporates these results. Potential areas of future research and implementation are also indicated, and a complete description of REVE's user commands is provided.

During the course of completing a system, the Knuth-Bendix completion procedure uses a reduction ordering to prove the termination of the rewriting system it constructs from the set of axioms. The choice of an appropriate ordering intimately depends on the particular axioms and the equations that get generated during the completion process. In the past, most Knuth-Bendix implementations have required that the reduction ordering be given *a priori*, a significant impediment to automatic theorem proving. Lescanne's REVE 1 introduced the important refinement of allowing, and helping, the user to dynamically extend the reduction ordering to order equations as they are encountered. REVE 2 improves on this scheme with the use of *automatic orderings*, whose implementations automatically compute all of the possible valid extensions to the ordering that allow an unorderable equation to be ordered. Here, we review the most popular classes of reduction orderings, present a new class of orderings that is more powerful than most, and present the theoretical and implementation issues in making these orderings automatic.

In addition, REVE 2 incorporates a new, "failure-resistant" implementation of the Knuth-Bendix completion procedure, which has been designed with automatic theorem proving in mind. This implementation uses a fine-grained approach to automatic equation postponement that categorizes equations based on the degree of difficulty they pose to the completion process. The Knuth-Bendix procedure is formulated as an ordered sequence of tasks designed to expedite the completion process and maximize the chances for successful termination. The order of the tasks within the sequence can be easily modified to accomodate varying requirements.

REVE is designed to be a practical, easy to use implementation of theoretical results pertaining to equational and inductive theorem proving using term rewriting. It has been carefully modularized and documented to facilitate understanding and use. REVE will have fulfilled its purpose if theoreticians can modify it to experiment with new results, and if software engineers can extend it for use in real world applications.

The organization of the thesis progresses from theory to practice. Chapter 2 is an introduction to equational and inductive theories, and proving theorems using rewriting systems and Knuth-Bendix. Chapter 3 introduces automatic orderings and presents a procedure for automatically constructing terminating rewriting systems. Chapter 4 describes the design of REVE's failure-resistant Knuth-Bendix implementation. Chapter 5 describes REVE itself: the user interface, example usage, and the program modules that comprise its CLU implementation. Chapter 6 summarizes the thesis, highlights some possible areas of future work, and reflects on the engineering obstacles encountered in building REVE. The Appendix describes, in detail, each of the user commands provided in the current version of REVE.

Chapter Two

Term Rewriting Systems and Proof Theory

2.1 Introduction

This chapter introduces equational theories, inductive theories, and term rewriting systems, as they pertain to REVE. We begin by defining notions related to terms and substitutions. We then discuss equational and inductive theories, and what it means to prove a theorem in each kind of theory. We describe term rewriting systems, and the process of rewriting. Two important properties of rewriting systems, termination and confluence, are characterized, and shown to provide a decision procedure for equational theories. We show how the Knuth-Bendix completion procedure can be used to generate such a decision procedure. Finally, we introduce inductionless induction, a technique that uses Knuth-Bendix to prove inductive theorems. Our development here takes an operational view of rewriting. See [Huet 80a] for a treatment using relations.

2.2 Terms and Substitutions

We assume a finite set of distinguishable symbols called *operators*. Examples of operators are + in arithmetic, *concat* in lists, and *true* in boolean. We also assume a disjoint set of distinguishable symbols called *variables*.

A *term* is defined inductively as either (1) a variable, or (2) an operator and a sequence of terms. In the latter case, if f is the operator and $t_1, ..., t_n$ is the sequence of terms, the term is denoted $f(t_1, ..., t_n)$, f is called the *root* operator, and the t_i are called the *arguments* of the term. The number of arguments, n, is called the *arity* of f. Here, we assume that an operator's arity is fixed. When the root is binary (i.e., has arity 2), we often use the infix form, e.g., $x + y$, and use parentheses to resolve ambiguity. An operator with zero arity is called a *constant*. We will denote a constant by its name, with no accompanying parentheses. We use $\mathcal{V}(t)$ to denote the set of variables that occur in a term t. When $\mathcal{V}(t) = \{\}$, t is said to be a *ground* term. By convention, we will reserve the symbols u, v, ..., z for variables, so that variables and constants can be distinguished.

The *subterms* of a term are the term itself and the subterms of its arguments. A subterm s within a term t can be designated by an *occurrence*, which is a sequence of positive integers denoting an access path in the term. We use Λ to denote the empty sequence. The *occurrence set* of a term t, $\mathcal{O}(t)$, is the set of occurrences of all its subterms. Formally,

$$\mathcal{O}(t) \equiv \{\Lambda\} \quad \text{if } t \text{ is a variable or constant,}$$
$$\mathcal{O}(t) \equiv \{\Lambda\} \cup \{i.q \mid q \in \mathcal{O}(t_i);\ i = 1, \ldots, n\} \quad \text{if } t = f(t_1, \ldots, t_n).$$

For example, if $t_1 = f(g(x), h(z)+1, y)$, $\mathcal{O}(t_1) = \{\Lambda, 1, 1.1, 2, 2.1, 2.1.1, 2.2, 3\}$. If $q \in \mathcal{O}(t)$, t/q is the subterm of the term t at occurrence q, defined by

$$t/\Lambda \equiv t,$$
$$t/i.q \equiv t_i/q \quad \text{if } t = f(t_1, \ldots, t_n).$$

For example, $t_1/2.1 = h(z)$. We use $t[q \leftarrow s]$ to denote the term t with the subterm at occurrence q replaced by the term s. Thus, $t_1[1.1 \leftarrow h(v)] = f(g(h(v)), h(z)+1, y))$.

A *substitution*, σ, is a mapping from variables to terms such that $\sigma(x) = x$ for all but a finite number of variables. We can represent a substitution by a finite set of ordered pairs, denoted $\sigma = \{x_1 \leftarrow t_1, \ldots, x_n \leftarrow t_n\}$. We extend the domain of a substitution to the set of all terms by defining

$$\sigma(f(t_1, \ldots, t_n)) = f(\sigma(t_1), \ldots, \sigma(t_n)).$$

For example, if we have the substitution $\sigma = \{x \leftarrow h(v), z \leftarrow g(g(z)), y \leftarrow z\}$ and the term $t_1 = f(z, g(y), v, h(x))$, we can apply σ to obtain $\sigma(t_1) = f(g(g(z)), g(z), v, h(h(v)))$.

Two terms, s and t, are said to be *unifiable* if and only if there exists a substitution, σ, such that $\sigma(s) = \sigma(t)$. The substitution σ is called a *unifier* of s and t. For example, if $s = f(g(x), h(y))$ and $t = f(y, z)$, one of their unifiers is $\sigma_1 = \{x \leftarrow 4+w, y \leftarrow g(4+w), z \leftarrow h(g(4+w))\}$. For this unifier, $\sigma_1(s) = \sigma_1(t) = f(g(4+w), h(g(4+w)))$. Whenever two terms are unifiable, they have a *most general unifier*, *mgu*, such that every unifier contains *mgu* as a factor (in terms of functional composition). The most general unifier of two terms is unique, up to variable renaming. For s and t above, $mgu = \{y \leftarrow g(x), z \leftarrow h(g(x))\}$. The unifier σ_1 above can be expressed as the functional composition $\sigma_1 = \sigma_2 \circ mgu$, where $\sigma_2 = \{x \leftarrow 4+w\}$. The *unification* of two terms, s and t, is $mgu(s)$ (which is the same as $mgu(t)$) for their most general unifier, *mgu*. With s and t as above, $f(g(x), h(g(x)))$ is their unification.

Unification plays a central role in resolution theorem proving [Robinson 65] and logic programming [Kowalski 74]. We shall use unification in the context of the Knuth-Bendix completion procedure, described in Section 2.6. Many algorithms to perform unification have been proposed (e.g., [Robinson 71] and [Baxter 73]), including those that run in linear-time [Paterson 78]. The algorithms in [Martelli 82] and [Corbin 83] are particularly fast in practice.

A term, s, is said to *match* (or *have the form of*) a term, t, if and only if there exists a substitution σ such that $s = \sigma(t)$. When the domain of σ is restricted to the set of variables in t, σ is unique and is called the *match of* s *by* t. For example, $s = f(g(h(y)), h(y))$ has the form of $t = f(g(x),x)$, and the match of s by t is $\sigma = \{x \leftarrow h(y)\}$. Matching can be thought of as "one-way" unification, where unification is permitted in only one of the terms.

2.3 Equations and Proof Theory

2.3.1 Equational Theories

An *equation* is an undirected pair of terms, written $s = t$. In equations, all variables are (implicitly) universally quantified. A *ground instance* of an equation, $s = t$, is an equation, $\sigma(s) = \sigma(t)$, that contains no variables, where σ is some substitution.

We are interested in the *equational theory*, $=_{\mathcal{E}}$, of a set of equations, $\mathcal{E}$. The *equational theory* of $\mathcal{E}$ consists of the closure of $\mathcal{E}$ under the following rules of inference: reflexivity, symmetry, transitivity, universal instantiation, and replacement of equals for equals. We say that $\mathcal{E}$ is a set of *axioms* for $=_{\mathcal{E}}$. If an equation, $s = t$, is in $=_{\mathcal{E}}$, we say that $s = t$ is an *equational theorem* (or *equational consequence*) of $\mathcal{E}$, and we write $s =_{\mathcal{E}} t$.

Figure 2-1 presents a set of axioms for groups. Here, $\bullet$ is the binary operation, x^{-1} denotes the inverse of x, and e is the identity. An example formal proof is given in Figure 2-2. Starting with the axioms, the rules of inference are used to prove that

$$(x^{-1})^{-1} = x \tag{1}$$

is an equational theorem. Note that the result of each proof step is itself an equational theorem.

Note that the group axioms, as given in Figure 2-1, state that e is the left identity, but not that it is the right identity. However, during the course of proving Equation 1 in Figure 2-2, we show, in Step 16, that $x \bullet e = x$ is an equational consequence of the axioms. The generation of useful "lemmas," such as this one, is also a by-product of the automatic theorem proving method in Section 2.6.

Figure 2-1: Axioms for Group Theory

(1) $e \bullet x = x$

(2) $x^{-1} \bullet x = e$

(3) $(x \bullet y) \bullet z = x \bullet (y \bullet z)$

Figure 2-2: Proof of an Equational Theorem About Groups

[1]	$e \bullet x = x$	(axiom)
[2]	$x^{-1} \bullet x = e$	(axiom)
[3]	$(x \bullet y) \bullet z = x \bullet (y \bullet z)$	(axiom)
[4]	$(x^{-1})^{-1} \bullet x^{-1} = e$	(apply $\sigma = \{x \leftarrow x^{-1}\}$ to [2])
[5]	$((x^{-1})^{-1} \bullet x^{-1}) \bullet x = x$	(insert [4] into [1])
[6]	$((x^{-1})^{-1} \bullet x^{-1}) \bullet x = (x^{-1})^{-1} \bullet (x^{-1} \bullet x)$	(apply $\sigma = \{x \leftarrow (x^{-1})^{-1}, y \leftarrow x^{-1}, z \leftarrow x\}$ to [3])
[7]	$((x^{-1})^{-1} \bullet x^{-1}) \bullet x = (x^{-1})^{-1} \bullet e$	(insert [2] into [6])
[8]	$(x^{-1})^{-1} \bullet e = x$	(insert [7] into [5])
[9]	$e \bullet e = e$	(apply $\sigma = \{x \leftarrow e\}$ to [1])
[10]	$(x^{-1})^{-1} \bullet (e \bullet e) = x$	(insert [9] into [8])
[11]	$((x^{-1})^{-1} \bullet e) \bullet e = (x^{-1})^{-1} \bullet (e \bullet e)$	(apply $\sigma = \{x \leftarrow (x^{-1})^{-1}, y \leftarrow e, z \leftarrow e\}$ to [3])
[12]	$((x^{-1})^{-1} \bullet e) \bullet e = x$	(insert [11] into [10])
[13]	$((x^{-1})^{-1} \bullet (x^{-1} \bullet x)) \bullet e = x$	(insert [2] into [12])
[14]	$(((x^{-1})^{-1} \bullet x^{-1}) \bullet x) \bullet e = x$	(insert [6] into [13])
[15]	$(e \bullet x) \bullet e = x$	(insert [4] into [14])
[16]	$x \bullet e = x$	(insert [1] into [15])
[17]	$(x^{-1})^{-1} \bullet e = (x^{-1})^{-1}$	(apply $\sigma = \{x \leftarrow (x^{-1})^{-1}\}$ to [16])
[18]	$(x^{-1})^{-1} = x$	(insert [8] into [17])

Most equational proofs, such as the one in Figure 2-2, are tedious and time-consuming to construct by hand. However, the myriad details in this style of proof are well-suited to automation.

2.3.2 Inductive Theories

Although "equational theory" is a useful notion in the context of algebraic structures like groups, it is less useful in the context of abstract data types. Consider the set of axioms about lists shown in Figure 2-3. These axioms presume that lists are built from the operators *null* (a constant) and *cons*, where *null* denotes the empty list, and where the first argument to *cons* is a list element and the second argument is a list. (For convenience, we use "list" here to mean a term that denotes a list.) The axioms describe *concat*, which concatenates two lists, and *reverse*, which reverses a list, in terms of *null* and *cons*. The equation

$$reverse(concat(cons(x, cons(y, null)), cons(z, null))) = cons(z, cons(y, cons(x, null)))$$

is an equational theorem of these axioms. However, most interesting and generally-applicable list theorems are not in the equational theory; e.g.,

$$reverse(reverse(x)) = x \qquad (2)$$

Figure 2-3: Axioms About the Theory of Lists

(1) $concat(null, x) = x$

(2) $concat(cons(x, y), z) = cons(x, concat(y, z))$

(3) $reverse(null) = null$

(4) $reverse(cons(x, y)) = concat(reverse(y), cons(x, null))$

Nevertheless, Equation 2 is a theorem, in the sense that every ground instance of Equation 2 that consists only of the operators in the axioms of Figure 2-3 is an equational theorem of those axioms. The *inductive theory* of a set of axioms consists of their equational theory, plus all equations for which all ground instances are in the equational theory[4]. We will refer to the equations in the inductive theory as *inductive theorems*. Below, we show that Equation 2 is an inductive theorem of lists.

The "inductive theory" is so named because we ordinarily prove inductive theorems using data type induction. This typically proceeds as follows: One designates certain operators as

[4]The initial algebra is a model of the inductive theory.

constructors of the data type of interest. One then shows that each ground term of that type is equivalent to at least one ground term consisting only of constructors; one usually proves this using induction on the structure of ground terms. If the latter property holds, the type is said to be *fully specified*[5] [Musser 80b]. Then, to prove an inductive theorem (again using structural induction), one need only show that the theorem holds for all ground terms consisting solely of constructors of the type.

Consider the constructors for lists. It can be shown that all ground terms constructable using the operators in Figure 2-3 can also be built using only *cons* and *null*. (We omit this proof here.) We designate *cons* and *null* to be the list constructors. Given the choice of operators in Figure 2-3, we have chosen the only minimal constructor set. In general, however, the minimal set will not always be unique. For example, if we define another operator, *append*, that appends an element to the right end of a list, {*cons*, *null*} and {*append*, *null*} serve equally well as minimal constructor sets, since any list can be constructed using the constructors in either set.

Having selected the constructor set {*null*, *cons*}, and proved (or asserted) that lists are fully specified with respect to these constructors, one may proceed to prove an inductive theorem. We first present a theorem that will be useful in our proof of Equation 2:

$$reverse(concat(x, cons(u, null))) = cons(u, reverse(x)) \tag{3}$$

A proof of this equation is given in Figure 2-4. In the proof, we induct over the number of elements in the list denoted by x; i.e., over the number of occurrences of *cons* in x. The basis step proves the theorem for lists with zero elements (i.e., *null* lists). The induction step assumes the induction hypothesis holds for lists of length n (we denote such lists by s), and proves the theorem for lists of length $n+1$ (denoted by $cons(v, s)$, where v is any list element). In this way, we prove the theorem for all lists, since any list can be constructed using *null* and *cons*. Using Equation 3 as a theorem, Figure 2-5 proves that Equation 2 is an inductive theorem of lists. Note that, except for the induction principle, a formal inductive proof uses the same rules of inference as in equational proofs.

Like equational proofs, proving inductive theorems is typically time-consuming. These proofs "by hand" also require creativity and trial-and-error to discover which inductive lemmas

[5]The notion of *full specification* is closely related to that of *sufficient completeness* [Guttag 78a].

Figure 2-4: Proof of an Inductive Lemma About Lists

[1] $\mathit{concat}(\mathit{null}, x) = x$ (axiom)
[2] $\mathit{concat}(\mathit{cons}(u, x), y) = \mathit{cons}(u, \mathit{concat}(x, y))$ (axiom)
[3] $\mathit{reverse}(\mathit{null}) = \mathit{null}$ (axiom)
[4] $\mathit{reverse}(\mathit{cons}(u, x)) = \mathit{concat}(\mathit{reverse}(x), \mathit{cons}(u, \mathit{null}))$ (axiom)

Basis step: Show that the theorem holds for the list *null*.
[5] $\mathit{reverse}(\mathit{cons}(u, \mathit{null})) = \mathit{concat}(\mathit{reverse}(\mathit{null}), \mathit{cons}(u, \mathit{null}))$ (apply $\sigma = \{x \leftarrow \mathit{null}\}$ to [4])
[6] $\mathit{concat}(\mathit{null}, \mathit{cons}(u, \mathit{null})) = \mathit{cons}(u, \mathit{null})$ (apply $\sigma = \{x \leftarrow \mathit{cons}(u, \mathit{null})\}$ to [1])
[7] $\mathit{reverse}(\mathit{concat}(\mathit{null}, \mathit{cons}(u, \mathit{null}))) = \mathit{concat}(\mathit{reverse}(\mathit{null}), \mathit{cons}(u, \mathit{null}))$
(insert [6] into [5])
[8] $\mathit{reverse}(\mathit{concat}(\mathit{null}, \mathit{cons}(u, \mathit{null}))) = \mathit{concat}(\mathit{null}, \mathit{cons}(u, \mathit{null}))$ (insert [3] into [7])
[9] $\mathit{reverse}(\mathit{concat}(\mathit{null}, \mathit{cons}(u, \mathit{null}))) = \mathit{cons}(u, \mathit{null})$ (insert [6] into [8])
[10] $\mathit{reverse}(\mathit{concat}(\mathit{null}, \mathit{cons}(u, \mathit{null}))) = \mathit{cons}(u, \mathit{reverse}(\mathit{null}))$ (insert [3] into [9])

Induction step: Assume the theorem holds for the list *s*. Show that it holds for the list *cons*(*v*, *s*).
[11] $\mathit{concat}(\mathit{cons}(u, \mathit{reverse}(s)), \mathit{cons}(v, \mathit{null})) = \mathit{cons}(u, \mathit{concat}(\mathit{reverse}(s), \mathit{cons}(v, \mathit{null})))$
(apply $\sigma = \{x \leftarrow \mathit{reverse}(s), y \leftarrow \mathit{cons}(v, \mathit{null})\}$ to [2])
[12] $\mathit{reverse}(\mathit{cons}(v, s)) = \mathit{concat}(\mathit{reverse}(s), \mathit{cons}(v, \mathit{null}))$ (apply $\sigma = \{x \leftarrow s, u \leftarrow v\}$ to [4])
[13] $\mathit{concat}(\mathit{cons}(u, \mathit{reverse}(s)), \mathit{cons}(v, \mathit{null})) = \mathit{cons}(u, \mathit{reverse}(\mathit{cons}(v, s)))$
(insert [12] into [11])
[14] $\mathit{concat}(\mathit{reverse}(\mathit{concat}(s, \mathit{cons}(u, \mathit{null}))), \mathit{cons}(v, \mathit{null})) = \mathit{cons}(u, \mathit{reverse}(\mathit{cons}(v, s)))$
(insert induction hypothesis into [13])
[15] $\mathit{reverse}(\mathit{cons}(v, \mathit{concat}(s, \mathit{cons}(u, \mathit{null})))) =$
$\mathit{concat}(\mathit{reverse}(\mathit{concat}(s, \mathit{cons}(u, \mathit{null}))), \mathit{cons}(v, \mathit{null}))$
(apply $\sigma = \{u \leftarrow v, x \leftarrow \mathit{concat}(s, \mathit{cons}(u, \mathit{null}))\}$ to [4])
[16] $\mathit{reverse}(\mathit{cons}(v, \mathit{concat}(s, \mathit{cons}(u, \mathit{null})))) = \mathit{cons}(u, \mathit{reverse}(\mathit{cons}(v, s)))$
(insert [15] into [14])
[17] $\mathit{concat}(\mathit{cons}(v, s), \mathit{cons}(u, \mathit{null})) = \mathit{cons}(v, \mathit{concat}(s, \mathit{cons}(u, \mathit{null})))$
(apply $\sigma = \{u \leftarrow v, y \leftarrow \mathit{cons}(u, \mathit{null}), x \leftarrow s\}$ to [2])
[18] $\mathit{reverse}(\mathit{concat}(\mathit{cons}(v, s), \mathit{cons}(u, \mathit{null}))) = \mathit{cons}(u, \mathit{reverse}(\mathit{cons}(v, s)))$
(insert [17] into [16])

Conclude:
[19] $\mathit{reverse}(\mathit{concat}(x, \mathit{cons}(u, \mathit{null}))) = \mathit{cons}(u, \mathit{reverse}(x))$
([10], [18], and induction principle)

Figure 2-5: Proof of an Inductive Theorem About Lists

We will use the fact that Equation 2-4 is an inductive theorem.

[1] *concat*(*null*, *x*) = *x* (axiom)
[2] *concat*(*cons*(*u*, *x*), *y*) = *cons*(*u*, *concat*(*x*, *y*)) (axiom)
[3] *reverse*(*null*) = *null* (axiom)
[4] *reverse*(*cons*(*u*, *x*)) = *concat*(*reverse*(*x*), *cons*(*u*, *null*)) (axiom)

Basis step: Show that the theorem holds for the list *null*.
[5] *reverse*(*reverse*(*null*)) = *null* (insert [3] into [3])

Induction step: Assume the theorem holds for the list *s*. Show that it holds for the list *cons*(*u*, *s*).
[6] *reverse*(*concat*(*reverse*(*s*), *cons*(*u*, *null*))) = *cons*(*u*, *reverse*(*reverse*(*s*))) (apply $\sigma = \{x \leftarrow reverse(s)\}$ to Equation 2-4)
[7] *reverse*(*concat*(*reverse*(*s*), *cons*(*u*, *null*))) = *cons*(*u*, *s*) (insert induction hypothesis into [6])
[8] *reverse*(*cons*(*u*, *s*)) = *concat*(*reverse*(*s*), *cons*(*u*, *null*)) (apply $\sigma = \{x \leftarrow s\}$ to [4])
[9] *reverse*(*reverse*(*cons*(*u*, *s*)) = *cons*(*u*, *s*) (insert [8] into [7])

Conclude:
[10] *reverse*(*reverse*(*x*)) = *x* ([5], [9], and induction principle)

should be proven before attempting to show the main theorem. In Section 2.7, we present a radically different, automatic method that can, in many cases, decide the validity or invalidity of equations with respect to the inductive theory. When applied to the problem of proving Equation 2, the method automatically "discovers" Equation 3 and proves it to be a theorem before proving Equation 2.

2.4 Term Rewriting Systems

Term rewriting systems are an important means for proving theorems in equational and inductive theories, and this is the use that concerns us here. Their mathematical properties also make them attractive as a model of computation; see [Dershowitz 83a] and [Goguen 79] for examples of these applications.

A *rewrite rule* (or, just *rule*) is a directed pair of terms, written $\lambda \rightarrow \rho$, such that every variable that occurs in ρ also occurs in λ. One may *reduce* (or *rewrite*) a term t using a rewrite rule $\lambda \rightarrow \rho$ if there is an occurrence $q \in \mathcal{O}(t)$ such that t/q matches λ. The reduced (rewritten) form of t is $t[q \leftarrow \sigma(\rho)]$, where σ is the match of t/q by λ. For example, if we have $(\lambda \rightarrow \rho) = f(g(y, y), x) \rightarrow g(x, x)$ and $t = f(f(g(a, a), h(y)), z)$, $t/1$ matches λ, $\sigma = \{y \leftarrow a, x \leftarrow h(y)\}$, and t is reduced to $f(g(h(y), h(y)), z)$. In this example, we can again use $\lambda \rightarrow \rho$ to reduce the resulting term and obtain $g(z, z)$.

A *term rewriting system* (or, just *rewriting system*) $\mathcal{R}$ is a finite set of rewrite rules. We write $s \rightarrow_{\mathcal{R}} t$ if and only if s can be reduced to t using one of the rewrite rules in $\mathcal{R}$ exactly once. The $\mathcal{R}$ subscript on $\rightarrow$ will be omitted when $\mathcal{R}$ is clear from context. The notation $s \rightarrow^*_{\mathcal{R}} t$ means that t can be obtained from s by applying rules from $\mathcal{R}$ zero or more times. We say that two terms s and s' are *joinable* if and only if there exists a term t such that $s \rightarrow^* t$ and $s' \rightarrow^* t$. When there exist zero or more terms $s_1, \ldots, s_n$ such that $t \rightleftarrows s_1 \rightleftarrows \ldots \rightleftarrows s_n \rightleftarrows t'$, where $\rightleftarrows$ denotes ($\rightarrow$ or $\leftarrow$), we write $t \leftrightarrow^*_{\mathcal{R}} t'$. For example, with $\mathcal{R} = \{a \rightarrow b, g(a, x) \rightarrow f(x, x)\}$, we have $f(a, a) \leftrightarrow^* g(b, b)$, since $f(a, a) \rightarrow f(a, b) \rightarrow f(b, b) \leftarrow g(a, b) \rightarrow g(b, b)$.

The equational theory of a rewriting system $\mathcal{R}$, denoted $=_{\mathcal{R}}$, is the equational theory of $\mathcal{R}$ when viewed as a set of equations. We can obtain a rewriting system $\mathcal{R}$ from a set of equations $\mathcal{E}$ using the following technique, suggested in [Knuth 70] and [Huet 80a]: For every equation $s = t$ in $\mathcal{E}$, choose nondeterministically one of the following:

(1) If $\mathcal{V}(s) \subseteq \mathcal{V}(t)$, put $t \rightarrow s$ in $\mathcal{R}$.

(2) if $\mathcal{V}(t) \subseteq \mathcal{V}(s)$, put $s \rightarrow t$ in $\mathcal{R}$.

(3) Let $X = \mathcal{V}(s) \cap \mathcal{V}(t) = \{x_1, \ldots, x_n\}$. Introduce a new operator f that does not appear in $\mathcal{E}$ or $\mathcal{R}$, and put the two rules $s \rightarrow f(x_1, \ldots, x_n)$ and $t \rightarrow f(x_1, \ldots, x_n)$ into $\mathcal{R}$.

The resulting rewriting system $\mathcal{R}$ will have the same equational theory as $\mathcal{E}$, except for the possible presence of new operators. If either of the first two actions above applies to $s = t$, we say that the equation is *compatible*. If only the third action applies, we say it is *incompatible*.

We will refer to the third action above as *dividing* an equation. Any equation may be divided, because since $s = t$ holds for all substitutions, its validity is independent of the values of variables not in X. Because the choice of action is nondeterministic, there may be more than one action that could apply to a given equation. For example, if an equation in $\mathcal{E}$ can be viewed as a rewrite rule in both directions, it can be placed into $\mathcal{R}$ either way, or it can be divided.

We have seen that one can generate a rewriting system, $\mathfrak{R}$, from a set of axioms, $\mathcal{E}$, such that $s =_{\mathcal{E}} t$ if and only if $s =_{\mathfrak{R}} t$ for all terms s and t. It can be easily shown that $s =_{\mathfrak{R}} t$ if and only if $s \leftrightarrow^*_{\mathfrak{R}} t$. Thus, if we have a decision procedure for $\leftrightarrow^*_{\mathfrak{R}}$, we have a decision procedure for the equational theory of $\mathcal{E}$. The next section describes two properties that, if they hold for $\mathfrak{R}$, let us decide $\leftrightarrow^*_{\mathfrak{R}}$.

2.5 Termination and Confluence

We say that a rewriting system, $\mathfrak{R}$, *terminates* (or that it is *noetherian*, *finitely terminating*, or *uniformly terminating*) if and only if there is no term t_1 for which there exists an infinite sequence of reductions $t_1 \rightarrow t_2 \rightarrow t_3 \rightarrow \ldots$. A term is *irreducible* if and only if it cannot be reduced by $\mathfrak{R}$. If $\mathfrak{R}$ terminates, any term, t, has at least one *normal form*, defined to be an irreducible term, $t\downarrow$, such that $t \rightarrow^* t\downarrow$. The rewriting system $\mathfrak{R} = \{((x+y)+z) \rightarrow (x+(y+z))\}$ terminates. However, the rewriting system $\mathfrak{R} = \{(x+y) \rightarrow (y+x)\}$ does not terminate, because we have $(a+b) \rightarrow (b+a) \rightarrow (a+b) \rightarrow \ldots$.

It is undecidable whether an arbitrary rewriting system terminates [Huet 78]. However, a number of methods have been proposed that prove termination in particular cases (see [Iturriaga 67], [Knuth 70], [Manna 70], [Lankford 75a], [Lipton 77], [Plaisted 78a], [Plaisted 78b], [Dershowitz 79a], [Lankford 79a], [Dershowitz 82a], [Guttag 83a], [Jouannaud 82a]). The most popular method, employed in REVE and described in Chapter 3, uses a *reduction ordering*, defined to be any well-founded partial ordering, $\succ$, on terms, such that $s \succ t \Rightarrow f(\ldots s \ldots) \succ f(\ldots t \ldots)$ and $\sigma(s) \succ \sigma(t)$ for any terms $f(\ldots s \ldots)$ and $f(\ldots t \ldots)$ and any substitution σ [Manna 70]. The termination proof consists of showing that $\lambda \succ \rho$ for every rule, $\lambda \rightarrow \rho$, in $\mathfrak{R}$.

Another important property for term rewriting systems is *confluence*. A rewriting system, $\mathfrak{R}$, is *confluent* (or *uniformly confluent* or *Church-Rosser*) if and only if, for all terms t, s, and s', $t \rightarrow^* s$ and $t \rightarrow^* s'$ implies s and s' are joinable. $\mathfrak{R}$ is said to be *convergent* (or *canonical* or *complete*) if it is both terminating and confluent.

When a rewriting system is confluent, the normal form of any term is unique, when the normal form exists. A sufficient condition for the existence of such a canonical form is the termination of all rewritings. Thus, for convergent rewriting systems, $\mathfrak{R}$, every term has a unique normal form. Furthermore, $\leftrightarrow^*$, and hence $=_{\mathfrak{R}}$ (see the last section), is decidable when $\mathfrak{R}$ is convergent: ($s =_{\mathfrak{R}} t$) if and only if ($s \leftrightarrow^* t$) if and only if ($s\downarrow\ = t\downarrow$). To test whether $s =_{\mathfrak{R}} t$, one can reduce both terms to normal form (by applying arbitrary reductions) and then check whether the normal forms are identical. Since $\mathfrak{R}$ terminates, this procedure is effective; reductions cannot continue indefinitely. The property of confluence is undecidable for an arbitrary term rewriting system, $\mathfrak{R}$. However, we will now see that one can decide confluence when $\mathfrak{R}$ terminates.

A rewriting system, $\mathfrak{R}$, is *locally confluent* if and only if, for all terms t, s, and s', $t \rightarrow s$ and $t \rightarrow s'$ implies s and s' are joinable. The definitions of confluence and local confluence differ in the number of reductions of t permitted to obtain s and s'. Note that confluence implies local confluence. The converse is not necessarily true. For instance, $\mathfrak{R} = \{a \rightarrow b, a \rightarrow c, b \rightarrow a, b \rightarrow d\}$ is locally confluent, even though a has two distinct normal forms, c and d. However, the following theorem is proved in [Newman 42]:

Theorem 1. A terminating rewriting system, $\mathfrak{R}$, is confluent if and only if it is locally confluent.

Similar "diamond lemmas" have been shown in [Knuth 70] and [Huet 80b]. It is difficult to test for local confluence as defined, since the definition quantifies over all terms. Theorem 1 is of interest to us only if it is easier to decide local confluence than confluence. This is indeed the case. We need the following definitions.

Two terms are said to *overlap* if and only if one is unifiable with a nonvariable subterm of the other, and the two terms share no variables. The *superposition* of two overlapping terms is the corresponding unification of one term and a subterm of the other term. To *superpose* two rewrite rules is to compute all of the superpositions between their left-hand sides. Let $\lambda_1 \rightarrow \rho_1$ and $\lambda_2 \rightarrow \rho_2$ be two rules in a rewriting system $\mathfrak{R}$ such that λ_1 and λ_2 overlap at occurrence q in λ_1, and let σ be the most general unifier of λ_1/q and λ_2. (We assume that variables have been renamed to alleviate sharing between the rules.) The *critical pair* associated with this overlap is $\langle \sigma(\lambda_1[q \leftarrow \rho_2]), \sigma(\rho_1) \rangle$. It consists of the two reductions of $\sigma(\lambda_1)$ by the two rules. Intuitively, a critical pair captures the way in which two rewrite rules might be used to rewrite a term into two different terms. For example, consider the two rules $f(x, g(x, h(y))) \rightarrow k(x, y)$ and $g(a, z) \rightarrow m(z)$. We can superpose the first rule at occurrence 2 with the second one, using the most general unifier $\{x \leftarrow a, z \leftarrow h(y)\}$, to obtain the critical pair $\langle f(a, m(h(y))), k(a, y) \rangle$. We will write a critical pair, $\langle s, t \rangle$, as an equation, $s = t$.

For a finite rewriting system $\mathcal{R}$, there are finitely many critical pairs. They can be effectively computed with the use of a unification algorithm. Their utility is apparent in the following theorem.

Theorem 2. A rewriting system, $\mathcal{R}$, is locally confluent if and only if every critical pair in $\mathcal{R}$ is joinable.

The original version of this theorem is presented in [Knuth 70], where it is used in conjunction with Theorem 1. Our statement of the theorem is from [Huet 80b], and does not require termination.

Combining Theorems 1 and 2 gives us a decision procedure for the confluence of terminating rewriting systems. The usefulness of convergent rewriting systems has already been argued. Note the dual importance of termination: it permits the use of Theorem 2 as a test for the confluence of a rewriting system $\mathcal{R}$, and helps us decide $\xrightarrow{*}_{\mathcal{R}}$ (and $=_{\mathcal{R}}$), when $\mathcal{R}$ is confluent, by alleviating infinite reductions. In some cases, a terminating, non-confluent rewriting system can be "completed" to produce a convergent rewriting system having the same equational theory. This is the subject of the next section.

2.6 The Knuth-Bendix Completion Procedure

Suppose we have a rewriting system $\mathcal{R}$, and a reduction ordering, $\succ$, such that $\lambda \succ \rho$ for all rules, $\lambda \rightarrow \rho$, in $\mathcal{R}$. By Theorem 2, we may test for local confluence by checking that all critical pairs are joinable. The two terms comprising a critical pair, $s = t$, are merely the result of reducing a single term by two different rewrite rules in $\mathcal{R}$, after applying a substitution. Consequently, $s = t$ is in $=_{\mathcal{R}}$, and $s \rightarrow t$ or $t \rightarrow s$ may be added to $\mathcal{R}$ without changing $=_{\mathcal{R}}$. Furthermore, if the two sides of the added rule are ordered under $\succ$ in the appropriate direction, the termination of $\mathcal{R}$ is preserved. Thus, if the local confluence test fails, i.e., if a non-joinable critical pair is found, and the critical pair is orderable, we may add the critical pair to $\mathcal{R}$, and test again for local confluence. If this process eventually causes $\mathcal{R}$ to be locally confluent, and no unorderable critical pairs were found, the resulting rewriting system is convergent, and has the same equational theory as the original.

The above method for "completing" $\mathcal{R}$ is the basis for the *Knuth-Bendix completion procedure*. The procedure, as originally described in [Knuth 70], is given in Figure 2-7. It incorporates the additional refinement that all rewrite rules are kept in normal form, for reasons of efficiency. In this figure, **repeat** means "go to the first statement of the smallest enclosing loop." Figure 2-7 makes use of the functions in Figure 2-6. The initial input to the procedure is a reduction ordering, $\succ$, and a (finite) rewriting system $\mathfrak{R}$, where $\forall\ \lambda \to \rho \in \mathfrak{R}$: $\lambda \succ \rho$. Later formulations of Knuth-Bendix accept equations as input and explicitly order these original equations into rules using $\succ$. REVE's implementation of Knuth-Bendix is described in Chapter 4.

Figure 2-6: Auxiliary Functions Used by Figure 2-7

Normal(t, R)	$\equiv$ A normal form of the term t with respect to the rewriting system R
Unorderable($s = t$)	$\equiv$ ($s \not\succ t$) **and** ($t \not\succ s$)
Order($s = t$)	$\equiv$ **if** $s \succ t$ **then** $s \to t$ **else** $t \to s$
CriticalPairs(r, r')	$\equiv$ All critical pairs between the rules r and r'

In looking for a decision procedure for an equational theory, $=_{\mathcal{E}}$, using Knuth-Bendix, one first selects a reduction ordering, $\succ$, and constructs a rewriting system, $\mathfrak{R}$, that consists of the axioms in $\mathcal{E}$, ordered, such that $\lambda \succ \rho$ for every rule, $\lambda \to \rho$, in $\mathfrak{R}$. One then executes the procedure in Figure 2-7. Knuth-Bendix is not an algorithm, in that it may halt in "failure" if the two sides of a rule are not orderable, or fail to terminate because it may generate an infinite set of rules. Consequently, any practical implementation needs to provide a means for stopping the main loop, perhaps by setting a limit on the number of iterations.

When $\succ$ is unable to order the two sides of a rewrite rule, it is either because $\succ$ is not general enough to show that $\mathfrak{R}$ terminates, or the rule is inherently non-terminating (e.g., $x + y \to y + x$). This is one of the major drawbacks of Knuth-Bendix as presented here: it does not apply to theories containing such (useful) permutative equations. The procedure can be extended, however, to work with certain equational theories with permutative axioms; see Section 6.2.4.3 for an overview of these results.

The Knuth-Bendix procedure has been successfully used on a number of interesting axiom

Figure 2-7: Description of the Original Knuth-Bendix Completion Procedure

```
Complete the initial system $\mathfrak{R}$:
loop

        Find non-joinable critical pair:
        for each $\lambda \rightarrow \rho$ in $\mathfrak{R}$ do
                for each $\gamma \rightarrow \mu$ in $\mathfrak{R}$ do
                        for each $s = t$ in CriticalPairs($\lambda \rightarrow \rho, \gamma \rightarrow \mu$) do
                                $s' :=$ Normal($s, \mathfrak{R}$); $t' :=$ Normal($t, \mathfrak{R}$)
                                if $s' \neq t'$ then goto Order equation endif
                        endfor
                endfor
        endfor
        halt with success

        Order equation:
        if Unorderable($s' = t'$) then halt with failure endif
        $(\lambda \rightarrow \rho) :=$ Order($s' = t'$)
        $\mathfrak{R} := \mathfrak{R} \cup \{\lambda \rightarrow \rho\}$

        Normalize rewriting system:
        for each $\gamma \rightarrow \mu$ in $\mathfrak{R}$ do
                $\gamma' :=$ Normal($\gamma, \mathfrak{R}$); $\mu' :=$ Normal($\mu, \mathfrak{R}$)
                if ($\gamma = \gamma'$) and ($\mu = \mu'$) then repeat endif
                if Unorderable($\gamma' = \mu'$) then halt with failure endif
                $\mathfrak{R} := (\mathfrak{R} - \{\gamma \rightarrow \mu\}) \cup \{$Order($\gamma' = \mu'$)$\}$
        endfor
endloop.
```

sets. One easy example is the *central groupoid* [Evans 67], which consists of one binary operator, •, and the single axiom

(1) $(x \bullet y) \bullet (y \bullet z) = y$

As shown by REVE, and indicated in [Knuth 70] and [Hullot 80a], the completed rewriting system consists of the above equation (ordered) plus the following two rewrite rules:

(2) $(x \bullet ((x \bullet y) \bullet z)) \rightarrow (x \bullet y)$

(3) $((x \bullet (y \bullet z)) \bullet z) \rightarrow (y \bullet z)$

This example can also be easily worked by hand. The latter two rules are the two critical pairs that result from overlapping the first axiom with itself.

A particularly interesting, and often referenced, example of an axiom set completable by Knuth-Bendix is group theory, as defined in Figure 2-1 on Page 19. Knuth-Bendix produces the convergent rewriting system shown in Figure 2-8. (There may be more than one convergent rewriting system corresponding to a set of axioms. Knuth-Bendix produces a different rewriting system for this example if the third equation in Figure 2-1 is ordered in the reverse direction.) The first three rules in Figure 2-8 are the original axioms. Note that the equational theorem $(x^{-1})^{-1} = x$, which we proved by hand in Figure 2-2, was explicitly generated as Rule 6 in Figure 2-8.

Figure 2-8: Convergent Rewriting System for Group Theory

(1) $e \bullet x \rightarrow x$

(2) $x^{-1} \bullet x \rightarrow e$

(3) $(x \bullet y) \bullet z \rightarrow x \bullet (y \bullet z)$

(4) $x^{-1} \bullet (x \bullet z) \rightarrow z$

(5) $e^{-1} \rightarrow e$

(6) $(x^{-1})^{-1} \rightarrow x$

(7) $x \bullet e \rightarrow x$

(8) $x \bullet x^{-1} \rightarrow e$

(9) $x \bullet (x^{-1} \bullet z) \rightarrow z$

(10) $(x \bullet y)^{-1} \rightarrow y^{-1} \bullet x^{-1}$

To prove, for example, the equational theorem

$$(x^{-1} \bullet y^{-1})^{-1} = y \bullet (x^{-1} \bullet e)^{-1} \qquad (4)$$

using the rewriting system in Figure 2-8, we reduce the left-hand side of the equation to normal form:

$(x^{-1} \bullet y^{-1})^{-1}$
$(y^{-1})^{-1} \bullet (x^{-1})^{-1}$ (apply Rule 10)
$y \bullet (x^{-1})^{-1}$ (apply Rule 6)
$y \bullet x$ (apply Rule 6 again)

and the right-hand side to normal form:

$y \bullet (x^{-1} \bullet e)^{-1}$
$y \bullet (e^{-1} \bullet (x^{-1})^{-1})$ (apply Rule 10)
$y \bullet (e \bullet (x^{-1})^{-1})$ (apply Rule 5)
$y \bullet (x^{-1})^{-1}$ (apply Rule 1)
$y \bullet x$ (apply Rule 6)

and note that the two normal forms are identical. If the normal forms were not identical, Equation 4 would not be an equational theorem of groups. See [Hullot 80a] for other examples of convergent rewriting systems.

In addition to its use as a means of obtaining decision procedures for equational theories, Knuth-Bendix may be used, among other purposes, to prove theorems by refutation [Hsiang 82]; to perform "meta-unification" in certain equational theories [Fay 79, Lankford 79b, Hullot 80b]; to interpret, verify, and synthesize "rewrite programs" [Dershowitz 83a]; and to compute the congruence closure of a finite set of ground equations [Lankford 75b]. See [Dershowitz 83b] for a survey of these applications. It was announced in [Butler 80] and proven in [Dershowitz 82b] that, for a given reduction ordering $\succ$, there is at most one convergent rewriting system corresponding to an equational theory. Thus, Knuth-Bendix may sometimes be used to prove that two different axioms sets have the same equational theory, by completing the two sets and comparing the resulting rewriting systems for equality (modulo variable renaming). Knuth-Bendix may also be used to prove inductive theorems, as explained in the next section.

2.7 Inductionless Induction

Musser [Musser 80b] first suggested using Knuth-Bendix to prove theorems in the inductive theory of a set of equations, as an alternative to performing explicit induction by hand. This idea, dubbed *inductionless induction* by Lankford, was extended and/or simplified in [Goguen 80], [Huet 80a], [Huet 82], [Lankford 81], [Dershowitz 83b], and [Kapur 84b]. We present here the method of Huet-Hullot [Huet 82], and interpret it in the context of the following theorem:

Theorem 3. [Dershowitz 83b] Let $\mathcal{R}$ be a convergent rewriting system for a set of axioms, $\mathcal{E}$, where $\succ$ is a reduction ordering used to establish the termination of $\mathcal{R}$. Let $\mathcal{G}\mathcal{R}$ be the set of irreducible ground terms in $\mathcal{R}$. Let $\mathcal{H}$ be a set of equations to be shown as inductive theorems. An equation in $\mathcal{H}$ is not valid in the inductive theory of $\mathcal{E}$ if and only if running Knuth-Bendix on $\mathcal{R} \cup \mathcal{H}$ with $\succ$ results in a rule with a left-hand side that has an instance in $\mathcal{G}\mathcal{R}$. This, provided the procedure does not terminate in "failure."

This suggests the following method for proving that the equations in $\mathcal{H}$ are valid in the inductive theory of $\mathcal{E}$: Complete $\mathcal{E}$ using Knuth-Bendix. Add the equations in $\mathcal{H}$ to the system and continue the completion process. If an instance of a term in $\mathcal{G}\mathcal{R}$ appears on the left-hand side of a rule, some equation in $\mathcal{H}$ is not valid in $\mathcal{E}$. If this does not occur, and Knuth-Bendix completes successfully, all of the equations in $\mathcal{H}$ are inductive theorems in $\mathcal{E}$. If Knuth-Bendix terminates in "failure" or generates an infinite set of rules, the method gives us no information about the validity of the $\mathcal{H}$ equations in $\mathcal{E}$. The main difficulty in this scheme is determining $\mathcal{G}\mathcal{R}$ from a given $\mathcal{R}$. The remainder of this section presents the Huet-Hullot approach to this problem.

Let $\mathcal{C}$ denote a chosen set of operators found in $\mathcal{E}$. We refer to these operators as *HH-constructors*. Let $\mathcal{G}\mathcal{E}$ denote all ground terms consisting only of operators found in $\mathcal{E}$, and $\mathcal{G}\mathcal{C}$ denote all ground terms consisting only of operators in the set $\mathcal{C}$. Before running Knuth-Bendix on $\mathcal{R} \cup \mathcal{H}$, one checks that $\mathcal{R}$ satisfies the *principle of definition*: every term in $\mathcal{G}\mathcal{E}$ is $\mathcal{E}$-congruent to exactly one term in $\mathcal{G}\mathcal{C}$. This check is difficult (indeed, undecidable) because both $\mathcal{G}\mathcal{E}$ and $\mathcal{G}\mathcal{C}$ are often infinite.

The Knuth-Bendix procedure is modified so that when it considers an equation, $s = t$, where the normal form, s', of s is not identical to the normal form, t', of t, the algorithm in Figure 2-9 is executed. The first case in the algorithm is an optimization, valid when the principle of definition holds, that divides $s = t$ into several smaller equations to assist in the successful completion of Knuth-Bendix. The second, third, and fourth cases in the algorithm ensure that distinct terms in $\mathcal{G}\mathcal{C}$ are not reducible to one another. The last two cases guarantee that the terms in $\mathcal{G}\mathcal{C}$ are less, under $\succ$, than all other terms in $\mathcal{G}\mathcal{E}$. Thus, when $\mathcal{R}$ satisfies the principle of definition, the last five cases together ensure that $\mathcal{G}\mathcal{C}$ consists precisely of the irreducible ground terms in $\mathcal{R}$, so $\mathcal{G}\mathcal{C}$ is the set $\mathcal{G}\mathcal{R}$ that we seek. Furthermore, the algorithm will halt with "pseudo-inconsistency"[6] if and only if a rule would be generated whose left-hand side has an instance in $\mathfrak{GC}$. The principle of definition and Figure 2-9 together imply the conditions required by Theorem 3. REVE incorporates Figure 2-9 into Knuth-Bendix, but does not currently provide support for checking the principle of definition. This is undecidable in general, but efforts are underway to incorporate a useful check for sufficient conditions (see Section 6.2.4.5).

Figure 2-9: Huet and Hullot's Inductionless Induction Modification to Knuth-Bendix

```
case
    $s' = f(s_1, ..., s_n)$ and $t' = f(t_1, ..., t_n)$, with $f \in \mathfrak{C}$: $\mathfrak{E} := (\mathfrak{E} - \{s' = t'\}) \cup \{s_i = t_i\}$; repeat
    $s' = f(...)$ and $t'$ is a variable, with $f \in \mathfrak{C}$: halt with pseudo-inconsistency
    $s'$ is a variable and $t' = f(...)$, with $f \in \mathfrak{C}$: halt with pseudo-inconsistency
    $s' = f(...)$ and $t' = g(...)$, with $f \in \mathfrak{C}, g \in \mathfrak{C}$, and $f \neq g$: halt with pseudo-inconsistency
    $s' = f(...)$ and $t' = g(...)$, with $f \in \mathfrak{C}, g \notin \mathfrak{C}$, and $s' \succ t'$: halt with failure
    $s' = f(...)$ and $t' = g(...)$, with $f \notin \mathfrak{C}, g \in \mathfrak{C}$, and $t' \succ s'$: halt with failure
endcase
```

As an example, consider using inductionless induction to prove inductive theorems in the theory of lists, as defined in Figure 2-3 on Page 20. We first use Knuth-Bendix to complete the list axioms[7], given a suitable reduction ordering, $\succ$. Here, Knuth-Bendix finds no non-joinable critical pairs, so the resulting rewriting system $\mathfrak{R}$ just consists of the equations in Figure 2-3, ordered. We then designate *null* and *cons* as the HH-constructors, and check that $\mathfrak{R}$ satisfies the principle of definition. The principle does hold for $\mathfrak{R}$, since all ground terms are irreducible, and $\mathfrak{GE} = \mathfrak{GC}$ in this example. We then continue running Knuth-Bendix on $\mathfrak{R}$, together with the set $\mathfrak{H}$ consisting, say, only of Equation 2 on Page 20. Knuth-Bendix will complete successfully in this case, and happens to produce Equation 3 on Page 21 as a critical pair that appears in the final convergent rewriting system. Note that Equation 3 is not

[6]In [Huet 82], the authors use the word "disproof" rather than "pseudo-inconsistency."

[7]In the Huet-Hullot approach, if $\mathfrak{E}$ itself satisfies the principle of definition, it is not strictly necessary that one first run Knuth-Bendix on $\mathfrak{E}$ before adding the equations in $\mathfrak{H}$. In practice, however, it is customary to run Knuth-Bendix first. The principle of definition is easier to check for a convergent rewriting system than for an arbitrary set of equations.

in the equational theory of the original list axioms, but it is in the equation theory of the list axioms plus Equation 2. Because Knuth-Bendix does not halt with "failure" or "pseudo-inconsistency" in this case, we conclude that Equation 2 (and Equation 3) are in the inductive theory of the original list equations. These theorems may, in turn, allow us to prove other inductive theorems. (For further examples, see Section 5.3.)

Chapter Three

Automatic Construction of Terminating Rewriting Systems

3.1 Introduction

In this chapter, we present a new, totally automatic method for constructing a rewriting system from a set of equations, and proving that the rewriting system terminates. Previous techniques have either required user help in guiding the proof, or have been too restrictive to be generally applicable. The ability to prove termination automatically is an important requirement in applications where the theorem prover is to be embedded in a larger program, especially when term rewriting is not the principal function of that program. In most such programs, it would be inappropriate to expect users to be sufficiently fluent in rewriting system termination techniques to assist in the termination proof.

Termination is undecidable. Nevertheless, we are often interested in whether a rewriting system, $\mathcal{R}$, terminates, because (see Section 2.5):

- Termination allows one to decide whether $\mathcal{R}$ is confluent.
- If $\mathcal{R}$ is confluent, termination allows one to decide $=_{\mathcal{R}}$.
- If $\mathcal{R}$ is not confluent, termination allows the use of the Knuth-Bendix completion procedure to help achieve confluence.

Knuth-Bendix, as part of the completion process, constructs a terminating rewriting system from a set of equations with the use of a reduction ordering, $\succ$. The construction process consists of showing that every equation can be ordered, in one direction or the other, into a rewrite rule, $\lambda \rightarrow \rho$, such that $\lambda \succ \rho$. These ordered rewrite rules comprise $\mathcal{R}$, and $\succ$ proves that $\mathcal{R}$ terminates.

In the context of Knuth-Bendix, the problem of ensuring that $\mathcal{R}$ terminates reduces to the problem of choosing an appropriate reduction ordering, $\succ$, if such an ordering can be found for the example at hand. In particular, constructing $\mathcal{R}$ in an automatic fashion consists of automatically finding $\succ$. Selecting $\succ$ based on a static analysis of the equations is not sufficient, because the set of equations grows during the completion process. The selection of $\succ$ must proceed dynamically: whenever an equation is encountered that cannot be ordered by $\succ$, $\succ$ must be extended (if possible) so that it can order the new equation and also order all previously-ordered equations. This idea was pioneered by Lescanne in REVE 1 [Lescanne 83a]. In this chapter, we present an automatic, dynamic procedure for extending $\succ$, so that the construction of $\mathfrak{R}$ and its proof of termination proceed automatically. The procedure is sufficiently general to be effective in a wide variety of practical applications.

The remainder of this chapter presents the theoretical justification and algorithmic methods supporting the automatic procedure for extending $\succ$. Section 3.2 presents the basic definitions and theory behind the use of orderings in constructing terminating rewriting systems. Section 3.3 describes and generalizes some popular classes of orderings. Section 3.4 introduces a new class of orderings that is more powerful than most other known classes of orderings for termination proofs. Section 3.5 presents methods for dynamically extending these orderings. Finally, Section 3.6 describes the automatic extension procedure itself. Sections 3.2 and 3.6 may be read independently of the other three, to obtain an overview of the scheme for automatically constructing terminating rewriting systems, while skipping the details of the orderings themselves.

3.2 Ordering Definitions and Properties

3.2.1 Relations, Relationals, Mappings, and Orderings

This chapter is concerned with various binary relations. A *binary relation*, φ, is a set of ordered pairs of elements belonging to a *base set*, S. The notation $s \ \varphi \ t$ means $\langle s, t\rangle \in \varphi$. A relation, φ_1, is an *extension* of another relation, φ_2, if and only if $\varphi_1 \supseteq \varphi_2$. The extension is *strict* if and only if the containment is proper.

A *relation pair* is inductively defined to be a pair, $\langle\varphi_1,\varphi_2\rangle$, where φ_1 and φ_2 are either relations or relation pairs. The *base set* of $\langle\varphi_1,\varphi_2\rangle$ is the union of the base sets of φ_1 and φ_2. A relation pair, $\langle\varphi_1,\varphi_2\rangle$, is *empty* if and only if both φ_1 and φ_2 are empty. We say that $\langle\varphi_1,\varphi_2\rangle$ is an *extension* of $\langle\varphi_1',\varphi_2'\rangle$ if and only if φ_1 is an extension of φ_1' and φ_2 is an extension of φ_2'. The extension is *strict* if and only if either of the constituent extensions is strict.

A *relational* is a relation that is parameterized on another relation or relation pair. If Φ is a relational on φ, Φ is *monotonic* in φ if and only if extending φ extends $\Phi[\varphi]$. An *instantiation* of Φ is any relation $\Phi[\varphi]$ where Φ is defined on φ. If Φ_1 and Φ_2 are both relationals on φ, we say that Φ_1 is an *extension* of Φ_2 if and only if $\Phi_1[\varphi] \supseteq \Phi_2[\varphi]$ with any φ for which Φ_2 is defined. The extension is *strict* if and only if the containment is proper. We will usually just write the name of the relational, say Φ, rather than $\Phi[\varphi]$, since φ will usually be clear from context.

Given a *domain* set, D, and a *range* set, R, a *(partial) mapping*, μ, from D to R is a binary relation with base set $D \cup R$, where $\mu(d) = r$ (i.e., $d \ \mu \ r$) only if $d \in D$, $r \in R$, and, for every $d \in D$, there is at most one r satisfying $\mu(d) = r$. The mapping μ is *total* if and only if there is exactly one such r for every d. We say that μ is *total over T* if it is total when its base set is restricted to T.

A *quasi ordering*, $\succeq$, is a transitive, reflexive binary relation. The notation $s \simeq t$ means ($s \succeq t$ and $t \succeq s$), and $s \not\succeq t$ means $\langle s, t\rangle \notin \succeq$. We say that s and t are *comparable* under $\succeq$ if and only if $s \succeq t$ or $t \succeq s$.

A *partial ordering*, $\succ$, is a transitive, irreflexive binary relation. The notation $s \not\succ t$ means $\langle s, t\rangle \notin \succ$. We can obtain a partial ordering, $\succ$, from a quasi ordering, $\succeq$, by defining $s \succ t$ if and only if ($s \succeq t$ and $t \not\succeq s$). We say that a partial ordering, $\succ$, is *well-founded* if and only if it admits no infinite descending sequences $s_1 \succ s_2 \succ s_3 \succ \ldots$ of elements in its base set. An *ordering* is any quasi or partial ordering.

3.2.2 Simplification Orderings

[Dershowitz 82a] introduced a general class of partial orderings on terms, known as *simplification orderings*, and showed that simplification orderings can be straightforwardly used to prove the termination of rewriting systems.

Definition 4. A partial ordering, $\succ$, on terms is a *simplification ordering* if it possesses the following two properties:

Compatibility: $s \succ t \Rightarrow f(\ldots s \ldots) \succ f(\ldots t \ldots)$
Subterm: $f(\ldots t \ldots) \succ t$

for any terms s, t, $f(\ldots s \ldots)$, and $f(\ldots t \ldots)$.

Theorem 5. [Dershowitz 82a] A term rewriting system, $\mathfrak{R}$, terminates if there exists a simplification ordering, $\succ$, such that $\sigma(s) \succ \sigma(t)$ for all substitutions σ and all rules $s \rightarrow t$ in $\mathfrak{R}$.

Every simplification ordering $\succ$ considered here is *stable*; i.e., $s \succ t$ implies $\sigma(s) \succ \sigma(t)$, for all terms s and t, and all substitutions σ. Consequently, we may use a variant of the above theorem that is slightly less general.

Theorem 6. A term rewriting system, $\mathfrak{R}$, terminates if there exists a stable simplification ordering, $\succ$, such that $s \succ t$ for all rules $s \rightarrow t$ in $\mathfrak{R}$.

In the last chapter, we indicated that Knuth-Bendix uses reduction orderings to prove termination. Theorem 6 indicates that stable simplification orderings can be used instead. Stable simplification orderings are different from reduction orderings, because stable simplification orderings are not necessarily well-founded, and reduction orderings do not necessarily have the subterm property. However, [Dershowitz 82a] showed that when the base sets of the orderings are restricted to terms over a finite set of operators, such as the terms that comprise a (finite) rewriting system, these two classes of orderings are the same. The notion of simplification ordering was introduced because it is usually much easier to show that an ordering is a simplification ordering than to show it is a reduction ordering. In applications other than termination proofs, when a well-founded ordering is needed for terms over an infinite set of operators, one must separately show the well-foundedness of the simplification ordering. See [Dershowitz 83c] for techniques in constructing well-founded orderings, and for an overview of most known classes of simplification orderings, including some of those discussed here.

3.2.3 Registered and Automatic Orderings

Most classes of simplification orderings in popular use can be viewed as what we will call *registered orderings*. A *registered relation* is any relational, parameterized on a *registry*, that yields a relation over terms. A *registry*, $\langle \pi, \psi \rangle$, is any relation pair consisting of a *precedence*, π, and a *status map*, ψ, representing information about operators. A *registered ordering* is any *registered relation* whose every instantiation is a stable simplification ordering.

It is important to recognize that a registered ordering is a relational, so it is not an ordering: it is a class of orderings. We use this terminology to be consistent with the names of existing classes of simplification orderings, such as the recursive path ordering. We will use the symbol $\succ$ both for registered orderings and simplification orderings. The meaning of $\succ$ will be clear from context.

A *precedence*, $\pi = \langle \trianglerighteq, \not\lessgtr \rangle$, is a relation pair, where $\trianglerighteq$ and $\not\lessgtr$ are binary relations on operators. We say that $\langle \trianglerighteq, \not\lessgtr \rangle$ is *consistent* if and only if all of the following are true:

(1) The relation $\trianglerighteq$ is a quasi ordering.

(2) The relation $\not\lessgtr$ is irreflexive and symmetric.

(3) For any three operators, f, g, and h, where $f \trianglerighteq g$ and $g \trianglerighteq h$, if $f \not\lessgtr g$ or $g \not\lessgtr h$, then $f \not\lessgtr h$.

We say that f and g are *comparable* under $\langle \trianglerighteq, \not\lessgtr \rangle$ if and only if they are comparable under $\trianglerighteq$. We will use $f \blacktriangleright g$ as a shorthand for ($f \trianglerighteq g$ and $f \not\lessgtr g$), and $f \doteq g$ as a shorthand for ($f \trianglerighteq g$ and $g \trianglerighteq f$). Note that if $f \trianglerighteq g$, one may extend $\langle \trianglerighteq, \not\lessgtr \rangle$ with $f \not\lessgtr g$ or $g \trianglerighteq f$ to obtain $f \blacktriangleright g$ or $f \doteq g$, respectively. Also note that $\blacktriangleright$ is a partial ordering. We say that $\langle \trianglerighteq, \not\lessgtr \rangle$ is *total* if and only if, for all operators f and g in the base set, either $f \blacktriangleright g$, $g \blacktriangleright f$, or $f \doteq g$. The precedence $\langle \trianglerighteq, \not\lessgtr \rangle$ is *total over T* if it is total when its base set is restricted to T. We will usually just use π to denote a precedence, rather than $\langle \trianglerighteq, \not\lessgtr \rangle$.

A *status map*, ψ, is a binary relation that represents some auxiliary information used by registered orderings. We say that ψ is *consistent* if and only if it is a partial mapping from operators to *statuses*. A *status* can have the value *multiset*, denoted Ⓜ; *left-to-right*, denoted Ⓛ; or *right-to-left*, denoted Ⓡ. If an operator, f, is not in the domain of a status map, ψ, the "status" of f is said to be *undefined*, written $\psi(f) =$ Ⓤ. Loosely, $\psi(f) =$ Ⓜ means that, for a term, t, whose root is f, the ordering regards the arguments of t as a multiset, and the order of the arguments is ignored. When $\psi(f) =$ Ⓛ, the leftmost arguments of t are given more weight in the ordering. Similarly, $\psi(f) =$ Ⓡ indicates that the rightmost arguments are more important. If $\psi(f) =$ Ⓤ, f has not yet been assigned a particular status. We say that the status of f has been *set* if and only if $\psi(f) \neq$ Ⓤ. The two statuses Ⓛ and Ⓡ are *lexicographic* in that they imply a lexicographic comparison of argument lists. (Left-to-right and right-to-left are not the only lexicographic possibilities, but they are the most useful.) Two statuses are *incompatible* if one is Ⓜ and the other is lexicographic, and are *compatible* otherwise. In registered orderings, the status of an operator is irrelevant if its arity is less than two.

The operators in the base set of a registry, $\langle \pi, \psi \rangle$, are implicitly assumed to be restricted to those occurring in the (finite) term rewriting system of interest. We say that $\langle \pi, \psi \rangle$ is *total* if and only if both π and ψ are total. The registry is *total over T* if and only if both π and ψ are total over T. The registry is *consistent* if and only if both π and ψ are consistent, and for all operators f and g such that $f \doteq g$, $\psi(f)$ and $\psi(g)$ are compatible. Registered orderings are not defined for inconsistent registries, so implementations should take precautions to preserve the consistency of the registry. Unless stated otherwise, all registries considered here are assumed to be consistent. We will denote the contents of particular registries using braces, for convenience; e.g., $\{f \gtrdot g,\ \psi(f) = Ⓜ\}$. We will usually just use ρ to denote a registry, rather than $\langle \pi, \psi \rangle$ or $\langle \langle \succeq, \not\preceq \rangle, \psi \rangle$.

To construct a terminating rewriting system from a set of equations, $\mathcal{E}$, using a registered ordering, $\succ$, one must find a *terminating registry*: a registry that allows every equation in $\mathcal{E}$ to be ordered by $\succ$ in one direction or the other. Thus, for $\succ$ to be useful in constructing terminating rewriting systems automatically, it must be possible to dynamically extend $\succ$ by extending the registry when an equation that is unorderable (under the current registry) is found. It is essential that $\succ$ be monotonic in the registry, so that extending the registry does not change the ordering of previously-ordered equations under $\succ$ [Lescanne 83a]. Another important property of $\succ$ is its *extensibility*: the degree to which $\succ$ can be extended by extending the registry.

For unorderable equations, we seek *extenders*. An *extender* for $s\ \varphi\ t$ under the registry ρ, where φ is a registered relation, is an extension of ρ such that $s\ \varphi\ t$ under that registry extension. The registry ρ is itself an extender if we already have $s\ \varphi\ t$. An extender is *minimal* if and only if no proper subset of that extender is also an extender. A *complete extender set*, S, for $s\ \varphi\ t$ under ρ is a set of registries such that every registry in S is an extender for $s\ \varphi\ t$ under ρ, and every extender for $s\ \varphi\ t$ under ρ is an extension of at least one registry in S. A *minimal complete extender set*, $\mathfrak{X}_t^s(\varphi, \rho)$, is a complete set of extenders that contains no non-minimal extenders. We will usually just write $\mathfrak{X}(\varphi)$ rather than $\mathfrak{X}_t^s(\varphi, \rho)$ when s, t, and ρ are irrelevant or clear from context. Note that a complete extender set explicitly includes every minimal extender. Consequently, every non-minimal extender in a complete extender set is an extension of some other (minimal) extender in that same set. Thus, $\mathfrak{X}_t^s(\varphi, \rho)$ can be obtained from a complete extender set for $s\ \varphi\ t$ under ρ, S, by removing all extenders from S that are extensions of other extenders in S. We say that $\mathfrak{X}_t^s(\varphi, \rho)$ is the *minimal reduction* of S.

The ability to compute $\mathfrak{S}(\succ)$ is the key to our method for automatically constructing a terminating rewriting system from a set of equations, by automatically finding a terminating registry under $\succ$. An *automatic ordering* is an implementation of a registered ordering, $\succ$, that can compute $\mathfrak{S}(\succ)$ when two terms are unorderable. In Section 3.5, we will show how some registered orderings can be implemented as automatic orderings.

Some interesting questions that one can ask about a registered relational are:

- Is every instantiation a simplification ordering?
- Is every instantiation stable?
- Is every instantiation well-founded?
- Is it monotonic in the registry?
- How extensible is it?
- Can it be implemented as an automatic ordering?

We will consider these questions for the registered relationals we will describe.

3.3 Path and Decomposition Orderings

This section discusses two important categories of registered orderings, one based on a recursive path traversal of terms, and one based on a comparison of term decompositions. The *recursive path ordering with status* (RPOS) is a widely used registered ordering, because it is powerful and easy to understand. A newer registered ordering, the *recursive decomposition ordering with status* (RDOS), is more powerful than RPOS, and can help extend the registry when two terms are unorderable.

Although both RPOS and RDOS allow the precedence to be incrementally extended during the termination proof, neither of these registered orderings permits the status map to be incrementally extended. Thus, one must set the status map *a priori*, rather than allowing it to be extended appropriately to order unorderable equations as they are encountered. This is a significant shortcoming for automatic termination proofs. In addition, both RPOS and RDOS are somewhat inflexible with respect to incremental precedence extensions during the termination proof. However, by changing the definitions of RPOS and RDOS slightly, we can correct these deficiencies in extensibility. We refer to the modified, fully extensible versions of these orderings as the *extensible path ordering with status* (EPOS) and the *extensible decomposition ordering with status* (EDOS).

In the remainder of this section, we present those definitions and properties of the above orderings that will be needed in the rest of the chapter. We will fully define the path orderings, because the definitions will be used in Section 3.5. We will also discuss the essential features of the decomposition orderings, though we will not require the details here.

3.3.1 Path Orderings

To define RPOS, we first need two subsidiary relationals on collections of elements, and two subsidiary functions.

Intuitively, a *multiset* (or *bag*), s, on a quasi ordering, $\succeq$, is an unordered collection of elements, where s may contain multiple elements that are equivalent under $\simeq$. More formally, s is a mapping from the base set, S, of $\succeq$ onto the nonnegative integers, that associates, with each member of S, the number of elements to which it is $\simeq$ in the multiset. We use $\{s_1, ..., s_m\}$ to denote the multiset containing the (possibly duplicated) elements $s_1, ..., s_m$. $\mathcal{M}(S)$ denotes the set of all finite multisets on S.

Definition 7. [Huet 80a] Given a quasi ordering, $\succeq$, whose base set is S, and elements s and t of $\mathcal{M}(S)$, we obtain a relational, $\succeq_{\mathrm{m}}$, on $\mathcal{M}(S)$, by $s \succeq_{\mathrm{m}} t$ if and only if $(\forall x)([t(x) > s(x)] \Rightarrow (\exists y)([y \succ x] \wedge [s(y) > t(y)]))$. The instantiations of $\succeq_{\mathrm{m}}$ are quasi orderings, called the *multiset orderings*.

See [Jouannaud 82b] for properties of this ordering, a comparison of this ordering with other multiset orderings, and an efficient implementation.

We will write a sequence as $\langle s_1, ..., s_m \rangle$. $\mathcal{L}(S)$ denotes the set of all finite sequences on S.

Definition 8. Given a quasi ordering, $\succeq$, whose base set is S, and elements $s = \langle s_1, s_2, ..., s_m \rangle$ and $t = \langle t_1, t_2, ..., t_n \rangle$ of $\mathcal{L}(S)$, we obtain a relational, $\succeq_{\mathrm{l}}$, on $\mathcal{L}(S)$, by $s \succeq_{\mathrm{l}} t$ if and only if $n = 0$, or $n > 0$, $m > 0$, $s_1 \succeq t_1$, and $\langle s_2, ..., s_m \rangle \succeq_{\mathrm{l}} \langle t_2, ..., t_n \rangle$. The instantiations of $\succeq_{\mathrm{l}}$ are quasi orderings, called the *lexicographic orderings*.

The LexSequence function takes a lexicographic status and a term, and re-orders the term's arguments (if necessary) to be appropriate for the given status:

```
    LexSequence($\gamma$, $\langle t_1, \ldots, t_n \rangle$) $\equiv$ if $\gamma$ = Ⓡ then $\langle t_n, \ldots, t_1 \rangle$ else $\langle t_1, \ldots, t_n \rangle$ endif
```

CompareEquivalent, which makes use of LexSequence, is used to compare two terms whose roots are $\doteq$ in the precedence. The function takes two pairs, where each pair consists of a term and a status assignment, plus a partial ordering for comparing arguments. CompareEquivalent compares the two terms, using the ordering, under the assumption that the roots of the terms are $\doteq$, and treating the terms as though the root of each term has the status assignment that is paired with that term. (In defining RPOS, the status assignment paired with each term will be the same as the status of the root of the term, but this will not be the case when we use CompareEquivalent in defining EPOS, below.)

```
CompareEquivalent($\langle s = f(s_1, \ldots, s_m), \gamma_1 \rangle, \langle t = g(t_1, \ldots, t_n), \gamma_2 \rangle, \succ$) $\equiv$
    case
        ($\gamma_1$ = Ⓜ) and ($\gamma_2$ = Ⓜ): $\{s_1, \ldots, s_m\} \succeq_{\overline{m}} \{t_1, \ldots, t_n\}$

        $\gamma_1$ and $\gamma_2$ are both lexicographic:
            [LexSequence($\gamma_1$, $s$) $\succeq_{\overline{x}}$ LexSequence($\gamma_2$, $t$)] and $(\forall t_i)(s \succ t_i)$
    endcase
```

In effect, CompareEquivalent compares the arguments of *s* and *t* as multisets if the statuses are both multiset, and compares them lexicographically, from left-to-right and/or right-to-left, if the statuses are both lexicographic. In addition, with lexicographic comparisons, CompareEquivalent must ensure that *s* is greater than each argument of *t*, if *s* is to be greater than *t*. CompareEquivalent is not defined if the two statuses are incompatible or if either status is Ⓤ.

Kamin & Lévy's RPOS registered ordering [Kamin 80], $\succ^{R}$, is an extension of the *recursive path ordering* [Dershowitz 82a]. RPOS is monotonic in the precedence [Lescanne 83b], and an instantiation of RPOS is well-founded if and only if $\gg$ is well-founded. (The partial ordering $\gg$ will always be well-founded if its base set of operators is finite.) The following definition makes use of Definitions 7 and 8, and of CompareEquivalent.

Definition 9. The *recursive path ordering with status*[8] (RPOS), $\succ^{R}$, is a registered relational. The partial ordering $\succ^{R}[\rho]$ is induced by the quasi ordering $\succeq^{R}[\rho]$, where

[8]Kamin & Lévy did not use a formal notion of status map. Our use of status is adapted from Lescanne's REVE 1 and from [Lescanne 84].

$$s = f(s_1, \ldots, s_m) \succeq^{R}[\rho]\ g(t_1, \ldots, t_n) = t$$

is defined inductively as the union of the following three cases:

(1) $(\exists s_i)(s_i \succeq^{R}[\rho]\ t)$

(2) $(f \gtrdot g)$ and $(\forall t_i)(s \succ^{R}[\rho]\ t_i)$

(3) $(f \doteq g)$ and CompareEquivalent$(\langle s, \psi(f)\rangle, \langle t, \psi(g)\rangle, \succ^{R}[\rho])$

(Note that f and g might be the same operator.) The ordering is only defined for consistent registries. We lift $\succ^{R}$ to a stable ordering on terms with variables by treating variables as constants, where: 1) $x \doteq x$ and $\psi(x) =$ Ⓜ (any status would do) for all variables x; and 2) $\langle x, y\rangle \notin \gtrdot$ and $\langle x, y\rangle \notin \not\doteq$ for all distinct symbols, x and y, where x and/or y is a variable.

Theorem 10. Every instantiation of $\succ^{R}$ is a simplification ordering.

Proof. See [Kamin 80]. □

Lemma 11. $\succ^{R}$ is monotonic in the precedence.

Proof. Easy extension of the argument in [Lescanne 83b] for the recursive path ordering. □

One would like to initialize the status of all operators to Ⓤ, and then incrementally choose status assignments for operators while constructing the rewriting system, as needed. Unfortunately, $\succ^{R}$ is not defined for Ⓤ status. Some implementations of $\succ^{R}$ (e.g., RRL [Kapur 84a] and REVE 1 [Lescanne 83a]) initially assign Ⓜ status to all operators, and then incrementally change the status of some operators to be lexicographic to help order unorderable equations. However, this is not a sound termination proof method, because it can cause previously-ordered rewrite rules to become unorderable.

For example, suppose that the status of both f and g is initially Ⓜ, and we have previously placed $f \gtrdot g$ in the precedence to order some previous equation. We encounter the equation

$$g(f(x, y)) = f(y, x) \qquad (5)$$

and find that the left-hand side is already greater than the right-hand side under $\succ^{R}$, with the current registry. Hence, we convert the equation into a rewrite rule and add it to the rewriting system, $\mathfrak{R}$. Later, we decide to change the status of f from Ⓜ to Ⓛ, to allow some other equation to be ordered. Making this change causes Equation 5 to become unorderable. Moreover, no further extensions to the registry will order the equation. We now have an unorderable rewrite rule in $\mathcal{R}$, and we can no longer be certain that $\mathcal{R}$ terminates. There exist other examples to show that this problem would remain even if we used Ⓛ or Ⓡ as the default status, instead of Ⓜ. In effect, such implementations of $\succ^{B}$ are not monotonic in the status map. Thus, since $\succ^{B}$ is not defined for Ⓤ status, the status map may not be incrementally changed during the construction of the terminating rewriting system.

The $\succ^{B}$ ordering is also somewhat inflexible with respect to incremental precedence extensions. Suppose that, for some operators f and g, $f \succeq g$ is in π, and that neither $f \not\preceq g$ nor $g \succeq f$ is in π, so that we have neither $f \succ g$ nor $f \doteq g$. (If a precedence contains such a pair of operators, we say that the precedence is *uncommitted*.) In this case, $\succ^{B}$ does not use the precedence information about f and g. Instead, $\succ^{B}$ treats f and g as though they are incomparable under π (see Definition 8). Consider a rewrite rule, $f(\ldots) \to g(\ldots)$, that is not orderable under π, but is orderable from left to right if π is extended with $f \not\preceq g$, and is also orderable from left to right if π is extended with $g \succeq f$. These are the only two possible π extensions involving f and g that still result in a consistent precedence. If one uses $\succ^{B}$, one is forced to choose between these two precedence extensions, even though either extension will order the rule. Choosing one of the two extensions (unnecessarily) limits flexibility in extending the registry when considering later unorderable rules.

We propose two modifications to $\succ^{B}$ to make it fully extensible:

(1) We permit an operator to have Ⓤ status, and use this as the initial status assignment for all operators. When comparing two terms, s and t, whose roots are $\doteq$ in the precedence, and one or both roots have Ⓤ status, we define s to be greater than t if and only if s would be greater than t under any assignment of compatible status(es) to those root(s) that have Ⓤ status. Thus, if we later set the status of a Ⓤ root, s and t will still be ordered. We only allow ϕ to be extended; i.e., we do not allow the status of an operator to be changed once it has been set. Using this scheme, we have the flexibility of delaying status assignments until they are needed, yet we have an ordering that is monotonic in ϕ.

(2) We use the partial information, $f \succeq g$, in the precedence, even if we do not have $f \not\preceq g$ or $g \succeq f$. In this case, if f and g are the roots of s and t, respectively, we define s to be greater than t if s would be greater than t under both $f \not\preceq g$ and $g \succeq f$. Thus, when ordering some later equation, we may extend the precedence

in either of these ways, and s and t will still be ordered. This change preserves the monotonicity of the ordering with respect to π.

By making the above modifications to $\succ^{R}$, we obtain EPOS, denoted $\succ^{P}$.

To define EPOS formally, we first define two subsidiary functions, AllStatuses and CompareAll. AllStatuses takes a status and returns a set of statuses. CompareAll, which uses the CompareEquivalent function declared above, takes two terms and a partial ordering, and compares the arguments of those terms (using the ordering) under all possible compatible status assignments to the roots of those terms, assuming that the two roots are $\doteq$ in the precedence.

```
AllStatuses($\gamma$) $\equiv$ if $\gamma$ = Ⓤ then {Ⓜ, Ⓛ, Ⓡ} else {$\gamma$} endif

CompareAll($s = f(\ldots), t = g(\ldots), \succ$) $\equiv$
    ($\forall\langle\gamma_1, \gamma_2\rangle \in$[AllStatuses($\psi(f)$)$\times$AllStatuses($\psi(g)$)]: $\gamma_1$ and $\gamma_2$ are compatible)
        CompareEquivalent($\langle s, \gamma_1\rangle, \langle t, \gamma_2\rangle, \succ$)
```

Definition 12. The *extensible path ordering with status* (EPOS), $\succ^{P}$, is a registered relational. The partial ordering $\succ^{P}[\rho]$ is induced by the quasi ordering $\succeq^{P}[\rho]$, where

$$s = f(s_1, \ldots, s_m) \succeq^{P}[\rho] \; g(t_1, \ldots, t_n) = t$$

is defined inductively as the union of the following three cases:

(1) $(\exists s_i)(s_i \succeq^{P}[\rho] \; t)$

(2) $(f \gtrdot g)$ and $(\forall t_j)(s \succ^{P}[\rho] \; t_j)$

(3) $([f \doteq g]$ or $[(f \, \underline{\gtrdot} \, g)$ and $(\forall t_j)(s \succ^{P}[\rho] \; t_j)])$ and CompareAll$(s, t, \succ^{P}[\rho])$

Variables are handled in the same manner as for $\succ^{R}$.

In the definition of $\succ^{P}$, the treatment of Ⓤ is the conjunction of the treatment given to Ⓜ, Ⓛ, and Ⓡ, and the treatment of $\underline{\gtrdot}$ is the conjunction of the treatment given to $\gtrdot$ and $\doteq$. When the status map is total and the precedence is committed, $\succ^{P} = \succ^{R}$.

Let us consider an example that illustrates the extensibility of $\succ^{P}$. Suppose we wish to find a terminating registry for the equations shown in Figure 3-1, under $\succ^{P}$. We start with an empty registry (the precedence is empty, the status of all operators is Ⓤ).

(1) The first equation in the figure is not orderable with $\succ^{P}$ under an empty registry.

However, the equation is orderable into a rewrite rule from left to right if we set $\psi(f)$ = Ⓛ, or it can be ordered from right to left if we set $\psi(f)$ = Ⓡ. We arbitrarily choose $\psi(f)$ = Ⓛ.

(2) The second equation cannot be ordered from right to left under any extension to the current registry (nor under any registry). The equation may be ordered from left to right if we extend the registry with either $\psi(g)$ = Ⓜ or $f \succeq g$. The first choice offers greater flexibility for later extending the precedence, and the second offers greater flexibility for later extending the status map. We arbitrarily choose the second of these registry extensions.

(3) The third equation is not orderable from right to left under any registry. However, it is orderable from left to right if we commit the precedence by extending it with $g \succeq f$, so that $f \doteq g$. We do so, and the equation becomes ordered into a rewrite rule from left to right.

(4) The fourth equation is not orderable from left to right under any registry. However, it is orderable from right to left if we set $\psi(g)$ = Ⓡ, so we extend the registry accordingly.

The final, terminating registry is $\{f \doteq g$, $\psi(f)$ = Ⓛ, $\psi(g)$ = Ⓡ$\}$.

Figure 3-1: Example to Illustrate the Extensibility of EPOS

(1) $f(f(x, x), y) = f(x, f(x, y))$

(2) $f(g(y, x), y) = g(x, y)$

(3) $g(f(y, x), x) = f(x, y)$

(4) $g(g(x, x), y) = g(y, g(x, y))$

In the above example, we happened to make the right choices for extending the registry so that a terminating registry was produced. In general, one cannot tell that a particular extender will not work until it is found to prevent the ordering of some later equation. In the example, further experimentation would reveal that no choices for extending the registry, other than the ones made above, allow a terminating rewriting system to be constructed using $\overset{P}{\succ}$. See Section 3.6 for a discussion of recovering from bad extender choices.

In practice, the $\succ$ relation between operators contributes the most toward ordering terms.

The status map is of secondary importance, though it is essential for ordering certain important equations, such as $(x + y) + z = x + (y + z)$, which expresses the associativity of $+$. Although $\doteq$ was needed for the above example, this is rarely the case. An important example where its use is required, however, is in the Knuth-Bendix completion of the one-axiom characterization of groups [Lescanne 83a].

3.3.2 Decomposition Orderings

Lescanne has recently developed the *recursive decomposition ordering with status* (RDOS) [Lescanne 84], which is an extension of the *recursive decomposition ordering* [Jouannaud 82a]. Like RPOS, RDOS is monotonic in the precedence, every instantiation of RDOS is a stable simplification ordering, and an instantiation of RPOS is well-founded if and only if $\triangleright$ is well-founded. RDOS and RPOS yield the same ordering when the precedence is total. RDOS is a strict extension of RPOS when the precedence is not total.

In addition, RDOS is *incremental* [Jouannaud 82a] in that an implementation can easily give some help to the user for extending the precedence when two terms are not orderable. This help consists of a complete set of all pairs of operators that might make the terms orderable, if used to extend $\triangleright$. As described in [Jouannaud 82a], these suggestions are not extenders, *per se*, because they are only single pairs of operators and they only address the $\triangleright$ relation in the registry. Nevertheless, the suggestions produced by RDOS are helpful and important, because (as noted in the previous section) the $\triangleright$ relation is usually the most significant information in the registry, and the set of suggestions produced is usually small. The decomposition orderings are the first ones to provide an easy way to help the user extend the registry.

RDOS has the same two extensibility limitations as RPOS: 1) RDOS requires that the status map be total before the termination proof begins, and 2) RDOS cannot take advantage of the partial information in uncommitted precedences. Again, these problems are easily fixed. RDOS can be straightforwardly extended to allow Ⓤ status for operators, and to handle $f \trianglerighteq g$, in a manner very similar to the way we changed RPOS into EPOS above. We call this modification to RDOS the *extensible decomposition ordering with status* (EDOS), $\succ^{D}$.

We do not give the details of RDOS (or EDOS) here. We have mentioned the decomposition orderings because they are more powerful than the path orderings, and their ability to provide suggestions is the inspiration for the automatic orderings described here. In the next section, we introduce a new ordering that is more powerful than both $\succ^{D}$ and $\succ^{P}$.

3.4 Closure Ordering with Status

Plaisted has suggested the *closure ordering with status*[9] (COS), a registered ordering that is more powerful than both EPOS and EDOS. In this section, we describe COS, and show that every instantiation of COS is a stable simplification ordering, COS is monotonic in the registry, and an instantiation of COS is well-founded if and only if $\gg$ is well-founded.

The definition of COS makes use of two subsidiary registered orderings, $\succ_{\bar{a}}$ and $\succ_{\bar{c}}$, that are relationals on other registered orderings. For a given registry $\rho = \langle \pi, \psi \rangle$, let $\mathcal{P}^s_t(\pi)$ denote the set of all total extensions of π over all operators that appear in s and/or t. (Note that all precedences in $\mathcal{P}^s_t(\pi)$ are committed with respect to these operators.) Let $\mathcal{S}^s_t(\psi)$ denote the set of all total extensions of ψ over those same operators, and let $\mathcal{A}^s_t(\rho)$ denote the set of all total extensions of ρ over those operators.

Definition 13. Let $\succ$ be a registered ordering. We define the registered ordering, $\succ_{\bar{a}}$, such that $s \succ_{\bar{a}} t$ if and only if $(\forall\ \rho \in \mathcal{A}^s_t(\rho))(s \succ[\rho]\ t)$.

Theorem 14. If every instantiation of $\succ$ is a simplification ordering, the same is true for $\succ_{\bar{a}}$.

Proof. We must show that every instantiation of $\succ_{\bar{a}}$ is a partial ordering, is compatible, and has the subterm property, for any $\succ$ whose every instantiation also has these properties.
Compatibility: (By contradiction.) Suppose that every instantiation of $\succ$ is compatible, but that this is not true for $\succ_{\bar{a}}$. Then for some registry, ρ, and some terms, s, t, $f(...s...)$, and $f(....t...)$, we have $s \succ_{\bar{a}} t$ and $f(...s...) \not\succ_{\bar{a}} f(...t...)$. By Definition 13, we have $s \succ t$ and $f(...s...) \not\succ f(...t...)$ for some registry extension in $\mathcal{A}^s_t(\rho)$, which contradicts the supposition.
Subterm: (By contradiction.) Suppose that every instantiation of $\succ$ has the subterm property, but that this is not true for $\succ_{\bar{a}}$. Then for some registry, ρ, and some terms, t and

[9] As suggested by Plaisted, the closure ordering does not use status, but it is easy to extend his idea in this manner.

$f(\ldots t \ldots)$, we have $f(\ldots t \ldots) \not\succ_{\mathrm{a}} t$. By Definition 13, we have $f(\ldots t \ldots) \not\succ t$ for some registry extension in $\mathcal{A}^{s}_{t}(\rho)$, which contradicts the supposition.
The proofs of transitivity and irreflexivity are similar. □

Lemma 15. If every instantiation of $\succ$ is stable, the same is true for $\succ_{\mathrm{a}}$.

Proof. (By contradiction.) Suppose that every instantiation of $\succ$ is stable, but that this is not true for $\succ_{\mathrm{a}}$. Then for some registry, ρ, some terms, s and t, and some substitution, σ, we have $s \succ_{\mathrm{a}} t$ and $\sigma(s) \not\succ_{\mathrm{a}} \sigma(t)$. By Definition 13, we have $s \succ t$ and $\sigma(s) \not\succ \sigma(t)$ for some registry extension in $\mathcal{A}^{s}_{t}(\rho)$, which contradicts the stability of $\succ$. □

Lemma 16. $\succ_{\mathrm{a}}$ is monotonic in the registry.

Proof. (By contradiction.) We must show that $\succ_{\mathrm{a}}$ is monotonic in both π and ψ. Suppose $\succ_{\mathrm{a}}$ is not monotonic in π. Then for some precedences π_1 and π_2, where π_2 is an extension of π_1, and for some ψ, $s \succ_{\mathrm{a}} t$ under $\langle \pi_1, \psi \rangle$, but $s \not\succ_{\mathrm{a}} t$ under $\langle \pi_2, \psi \rangle$. By Definition 13, it must therefore be the case that $s \succ t$ under ψ and all precedences in $\mathcal{P}^{s}_{t}(\pi_1)$, but not under ψ and all precedences in $\mathcal{P}^{s}_{t}(\pi_2)$. But this is a contradiction, since $\mathcal{P}^{s}_{t}(\pi_1) \supseteq \mathcal{P}^{s}_{t}(\pi_2)$. The proof for ψ-monotonicity is similar. □

Lemma 17. Assume $\succ$ is monotonic in the registry. If an instantiation of $\succ$ is well-founded whenever $\gtrdot$ is well-founded, the same is true for $\succ_{\mathrm{a}}$. If $\succ_{\mathrm{a}}$ is well-founded under some registry, ρ, $\succ$ is also well-founded under ρ.

Proof. (By contradiction.) Suppose that an instantiation of $\succ$ is well-founded whenever $\gtrdot$ is well-founded, but that this is not true for $\succ_{\mathrm{a}}$. Then there exists an infinite decreasing sequence $t_1 \succ_{\mathrm{a}} t_2 \succ_{\mathrm{a}} t_3 \succ_{\mathrm{a}} \ldots$ for some precedence π for which $\gtrdot$ is well-founded, and for some ψ. By Definition 13, we have $t_1 \succ t_2 \succ t_3 \succ \ldots$ under all registries in $\mathcal{A}^{s}_{t}(\rho)$. Since $\gtrdot$ is well-founded, there is (by Zorn's Lemma) some total extension, $\pi_1 = \langle \gtrsim_1, \not\approx_1 \rangle$, of π, such that $\gtrdot_1$ is well-founded, and (by supposition) $\succ$ is well-founded under $\langle \pi_1, \psi \rangle$. Since $\succ$ is monotonic in ψ, $\succ$ is well-founded under all registries in $\{\pi_1\} \times \mathcal{S}^{s}_{t}(\psi)$. But this is a contradiction, since $\{\pi_1\} \times \mathcal{S}^{s}_{t}(\psi)$ is non-empty and $\mathcal{A}^{s}_{t}(\rho) \supseteq \{\pi_1\} \times \mathcal{S}^{s}_{t}(\psi)$, and $\succ$ is not well-founded under any registries in $\mathcal{A}^{s}_{t}(\rho)$.
Suppose $\succ_{\mathrm{a}}$ is well-founded under some registry, ρ, and $\succ$ is not. Then there exists an infinite decreasing sequence $t_1 \succ t_2 \succ t_3 \succ \ldots$ under all registries in $\mathcal{A}^{s}_{t}(\rho)$, since $\succ$ is monotonic in ρ. By Definition 13, we have $t_1 \succ_{\mathrm{a}} t_2 \succ_{\mathrm{a}} t_3 \succ_{\mathrm{a}} \ldots$ under ρ, which contradicts the supposition. □

Corollary. If an instantiation of $\succ$ is well-founded if and only if $\gg$ is well-founded, and $\succ$ is monotonic in the registry, then an instantiation of $\succ_{\mathfrak{a}}$ is well-founded if and only if $\gg$ is well-founded.

Definition 18. Given a registered ordering, $\succ$, we obtain its *closure*, $\succ_{\mathfrak{c}}$, where $s \succ_{\mathfrak{c}} t$ is defined as the union of the following two cases:

(1) $s \succ t$

(2) $s \succ_{\mathfrak{a}} t$

Lemma 19. If $\succ$ is monotonic in the registry, $\succ_{\mathfrak{c}} = \succ_{\mathfrak{a}}$.

Proof. Assume $\succ$ is monotonic in the registry. By Definitions 13 and 18, $s \succ_{\mathfrak{a}} t$ implies $s \succ_{\mathfrak{c}} t$, and (2) (above) implies $s \succ_{\mathfrak{a}} t$. We must also show that (1) (above) implies $s \succ_{\mathfrak{a}} t$. Suppose (1). Since $\succ$ is monotonic in the registry, we have $s \succ t$ under all registries in $\mathcal{A}^{s}_{t}(\rho)$. By Definition 13, $s \succ_{\mathfrak{a}} t$. $\square$

Note that the closure, $\succ_{\mathfrak{c}}$, of a registered ordering, $\succ$, is usually more efficient to compute than $\succ_{\mathfrak{a}}$ because, by the definition of $\succ_{\mathfrak{c}}$, $s \succ_{\mathfrak{a}} t$ need not be computed if we already have $s \succ t$. The closure operation unifies $\succeq^{P}$ and $\succeq^{p}$, in that the closure of $\succeq^{P}$ is the same registered ordering as the closure of $\succeq^{p}$. I.e.,

Lemma 20. $\succ^{P}_{\mathfrak{c}} = \succ^{p}_{\mathfrak{c}}$

Proof. When the registry is total, $\succeq^{P} = \succeq^{p}$ (see Section 3.3.2). Thus, by Definition 13, $\succ^{P}_{\mathfrak{a}} = \succ^{p}_{\mathfrak{a}}$. Since both $\succeq^{P}$ and $\succeq^{p}$ are monotonic in the registry, we have $\succ^{P}_{\mathfrak{c}} = \succ^{p}_{\mathfrak{c}}$, by Lemma 19. $\square$

For concreteness, we use $\succ^{P}_{\mathfrak{c}}$ instead of $\succ^{p}_{\mathfrak{c}}$, and obtain Plaisted's registered ordering[10].

Definition 21. The *closure ordering with status* (COS), $\succeq^{*}$, is the closure, $\succ^{P}_{\mathfrak{c}}$, of $\succeq^{P}$.

[10]Plaisted's definition of the closure ordering is more general than the one we give here. His definition treats variables as operators in the total precedences under which EPOS is computed. This results in a more powerful closure ordering. The proof of stability for this improved closure ordering is more complicated than for our definition, because it does not follow directly from the stability of $\succeq^{P}$. Such a proof would be a digression here, so we have presented the simpler definition. This improvement to the closure ordering is largely independent of the automatic termination issues discussed in this chapter.

We will write $\succ^{*}$ in place of $\succ^{P}_{c}$ in the remainder of this thesis. The monotonicity of $\succ^{P}$ implies that $\succ^{*} = \succ^{P}_{a}$, by Lemma 19. Consequently, Theorem 14, Lemmas 15 and 16, and the corollary to Lemma 17 apply to $\succ^{*}$ as well as $\succ^{P}_{a}$. Thus, since every instantiation of $\succ^{P}$ is a stable simplification ordering, $\succ^{P}$ is monotonic in the registry, and an instantiation of $\succ^{P}$ is well-founded if and only if $\triangleright$ is well-founded (see Section 3.3.1), these properties also hold for $\succ^{*}$. The next theorem states the main reason for our interest in $\succ^{*}$.

Theorem 22. $\succ^{*}$ is a strict extension of $\succ^{D}$.

Corollary. $\succ^{*}$ is a strict extension of $\succ^{P}$.

Proof. Assume $s \succ^{D} t$ under ρ. Then we have $s \succ^{D} t$ under all registries in $\mathcal{A}^{s}_{t}(\rho)$, because $\succ^{D}$ is monotonic in the registry. Since $\succ^{D} = \succ^{P}$ when the precedence is total, we have $s \succ^{P} t$ under all registries in $\mathcal{A}^{s}_{t}(\rho)$. By the monotonicity of $\succ^{P}$, Lemma 19, and Definitions 13 and 21, we have $s \succ^{*} t$ under ρ. Thus, $\succ^{*}$ is an extension of $\succ^{D}$.
To see that the extension is strict, consider the two terms,

$$s = f(f(a, a), f(b, b))$$
$$t = f(b, a)$$

Assume that the registry is empty. We have $s \succ^{*} t$, but s and t are not orderable under $\succ^{D}$. Since $\succ^{D}$ is a strict extension of $\succ^{P}$, the corollary follows immediately. □

To order the above two terms under $\succ^{D}$ and $\succ^{P}$, one can extend the registry in any of several ways, including $\psi(f) = \textcircled{M}$, or $a \trianglerighteq b$, or $b \trianglerighteq a$, or any registry extension in $\mathcal{A}^{s}_{t}(\rho)$, each of which causes s to be greater than t.

As an aside, the above example does not demonstrate the added power of $\succ^{*}$ over the recursive path ordering (RPO) [Dershowitz 82a]. RPO is the same as RPOS, except that the status of all operators is $\textcircled{M}$. Above, when $\psi(f) = \textcircled{M}$, we have $s \succ^{D} t$, as well as $s \succ^{*} t$. Lescanne has suggested another example:

$$s = f(f(f(a, a), a), f(b, b))$$
$$t = f(f(a, b), f(a, b))$$

Here, s and t are not orderable under $\succ^{D}$ when the precedence is empty and all operators have $\textcircled{M}$ status. However, we do have $s \succ^{*} t$ in this case.

On the face of it, $\succ^{*}$ looks to be a mixed blessing. On the one hand, the $\succ^{*}$ registered ordering is more powerful than $\succ^{D}$. On the other hand, a $\succ^{*}$ implementation based directly on the definition would run very slowly in the worst case. For two terms, *s* and *t*, under an empty registry, where $\langle s, t\rangle \notin \succ^{P}$, $s \succ^{*} t$, and *s* and *t* include 5 different operators (not atypical), it appears that there are $5! \times 5^3 = 15{,}000$ total registries under which $\succ^{P}$ must be computed in an attempt to order the equation under $\succ^{*}$. However, the next section presents a method of computing $\succ^{*}$ that may be more efficient.

3.5 Computing Minimal Extenders

As discussed in Section 3.2.3, it is highly desirable to compute the minimal complete extender set, $\mathfrak{E}(\succ)$, whenever two terms, *s* and *t*, are found to be unorderable under $\succ$. This section describes methods for computing $\mathfrak{E}(\succ^{P})$, $\mathfrak{E}(\succ^{D})$, and $\mathfrak{E}(\succ^{*})$, allowing $\succ^{P}$, $\succ^{D}$, and $\succ^{*}$ to be implemented as automatic orderings for automatic termination proofs. We show that $\mathfrak{E}(\succ^{*})$, and even $\succ^{*}$ itself, can be computed using either $\mathfrak{E}(\succ^{P})$ or $\mathfrak{E}(\succ^{D})$. The $\mathfrak{E}(\succ^{P})$ scheme has been implemented in REVE, and Lescanne is currently developing a $\mathfrak{E}(\succ^{D})$ implementation. Some further study is required before implementing a $\mathfrak{E}(\succ^{*})$ scheme.

The computing of minimal extenders has been largely ignored in the past. The precedence and status map are typically chosen *a priori*, and then appropriately adjusted in a trial-and-error fashion. There are three major reasons for this:

(1) Until recently, even manually-produced termination proofs have been difficult to obtain. Only in the last several years have classes of simplification orderings, such as RPOS and RDOS, emerged that are sufficiently general to be applicable to a wide variety of rewriting systems found in practice.

(2) Prior to the emergence of Lescanne's REVE 1, the idea of extending the registry on an as-needed basis, as unorderable equations are encountered, had not appeared in any available system.

(3) Minimal extenders seem computationally intractable. Any algorithm for computing $\mathfrak{E}(\succ)$ probably requires time that is exponential in the number of operators in the terms *s* and *t*.

A goal of this chapter is to pragmatically address the last concern above. The methods for computing minimal extenders that we present here have probable worst-case exponential behavior. However, for typical examples, we have found that the $\mathfrak{E}(\succ^{P})$ algorithm usually requires no more than several seconds per equation, and we conjecture that the running time of the $\mathfrak{E}(\succ^{D})$ and $\mathfrak{E}(\succ^{*})$ algorithms will be similar. Moreover, when constructing a terminating rewriting system from a typical set of equations, many of the equations will already be orderable under the current registry, and it is only necessary to compute $\mathcal{E}(\succ)$ when an equation is not orderable.

In the remainder of this section, we describe the $\mathcal{E}(\succ^{\underline{P}})$ computation in detail, briefly indicate the differences between computing $\mathcal{E}(\succ^{\underline{D}})$ and $\mathcal{E}(\succ^{\underline{P}})$, and give an overview of a technique for computing $\mathcal{E}(\succ^{\star})$. Throughout, we assume that the two terms being compared are $s = f(s_1, ..., s_m)$ and $t = g(t_1, ..., t_n)$, and that all variables are to be regarded as constants, as indicated above in the definitions of $\succ^{\underline{R}}$ and $\succ^{\underline{P}}$.

3.5.1 Minimal Extenders for EPOS and EDOS

This section presents the terminology, concepts, and algorithms related to the computing of $\mathcal{E}(\succ^{\underline{P}})$ and $\mathcal{E}(\succ^{\underline{D}})$. We will present the details of our algorithm for computing $\mathcal{E}(\succ^{\underline{P}})$. The method for computing $\mathcal{E}(\succ^{\underline{D}})$ is similar, so we will only indicate how the $\mathcal{E}(\succ^{\underline{D}})$ generation scheme differs from the one for $\mathcal{E}(\succ^{\underline{P}})$. For concreteness, all terminology will be introduced in the context of $\succ^{\underline{P}}$.

The $\mathcal{E}(\succ^{\underline{P}})$ algorithm makes use of *comparators* and *orderals*. We will see that the problem of computing the minimal extenders for $s \succ^{\underline{P}} t$ reduces to the problem of computing extenders for the orderals of $s \succ^{\underline{P}} t$ under each *incremental extension*. This, in turn, reduces to the problem of computing *combined extenders* for the orderals, which then reduces to the problem of computing the minimal extenders for the comparators that compose the orderals.

A *relator*, φ, is one of three registered relations used in defining and computing $\succ^{\underline{P}}$. The value of φ may be either $\succ^{\underline{P}}$, $\succeq^{\underline{P}}$, or $\simeq^{\underline{P}}$.

A *comparator*, denoted $\langle s,t,\varphi \rangle$, associates a particular pair of terms (here, s and t) with a relator (φ) under which they should be compared. A registry is a *(minimal) extender* for a comparator, $\langle s,t,\varphi \rangle$, if and only if it is a (minimal) extender for $s \ \varphi \ t$. The notions of *complete extender set*, *minimal complete extender set*, and *minimal reduction* (see Section 3.2.3) carry over straightforwardly to comparator extenders.

An *orderal*, D, for a comparator, $\langle s,t,\varphi \rangle$, under ρ is a set of comparators such that:

(1) For every comparator $\langle s',t',\varphi'\rangle$ in D, $s' \in \{s, s_1, ..., s_m\}$, $t' \in \{t, t_1, ..., t_n\}$, and it is not the case that both $s' = s$ and $t' = t$.

(2) If ρ is an extender for every comparator in D, it is an extender for $\langle s,t,\varphi\rangle$.

(3) No subset of D is an orderal.

Intuitively, the comparators in an orderal represent subterms that can be compared to establish $s\ \varphi\ t$. The orderals for $\langle s,t,\varphi\rangle$ are not defined if both s and t are constants. An *extender* for an orderal, D, is an extension of the current registry that is an extender for every comparator in D. An extender for D is *minimal* if no proper subset of that extender is also an extender for D. The notion of *complete extender set* carries over straightforwardly to orderal extenders. A *combined extender* for an orderal D is a union of extenders that consists of exactly one minimal extender from each comparator in D, provided that union results in a consistent registry.

We will use $\mathfrak{O}_t^s(\varphi,\rho)$ to denote the complete set of orderals for $\langle s,t,\varphi\rangle$ under ρ. The orderals in $\mathfrak{O}_t^s(\varphi,\rho)$ are derived directly from the definitions of $\succ^{\rho}$, $\succeq^{\rho}$, or $\simeq^{\rho}$, depending on φ. Consider the comparator $\langle s,t,\succ^{\rho}\rangle$. If $f \rhd g$, there is only one orderal: $\{\langle s,t_1,\succ^{\rho}\rangle, ..., \langle s,t_n,\succ^{\rho}\rangle\}$. If $\langle f, g\rangle \notin \unrhd$, there are m orderals for $\langle s,t,\succ^{\rho}\rangle$: $\{\langle s_1,t,\succeq^{\rho}\rangle\}, ..., \{\langle s_m,t,\succeq^{\rho}\rangle\}$. There are typically many orderals when $f \doteq g$, and so on. A *complete extender set* for $\mathfrak{O}_t^s(\varphi,\rho)$ is any set of registries that is a complete extender set for every orderal in $\mathfrak{O}_t^s(\varphi,\rho)$.

An *incremental extension* of $\rho = \langle\langle \unrhd, \not\doteq\rangle,\psi\rangle$ for s and t is any extension to ρ that differs from ρ only in that it may contain additional information about f and g. For example, if f and g are not comparable under $\langle \unrhd, \not\doteq\rangle$, and both have Ⓤ status, then ρ, $\rho \cup \{g \rhd f\}$, and $\rho \cup \{f \unrhd g, \psi(f) = \text{Ⓜ}\}$ are incremental extensions. The *incremental extension set*, denoted $\mathfrak{I}_t^s(\rho)$, of ρ for s and t is the set of all such incremental extensions.

Note that every minimal extender for an orderal, D, must also be a combined extender for D, and every combined extender for D is an extender for D. Thus, the set of all combined extenders for an orderal is a complete extender set for that orderal. Therefore, by the definition of $\mathfrak{O}_t^s(\varphi,\rho)$ complete extender sets, the set of all combined extenders for all orderals in $\mathfrak{O}_t^s(\varphi,\rho)$ is a complete extender set for $\mathfrak{O}_t^s(\varphi,\rho)$.

To compute the minimal complete extender set for some comparator $\langle s,t,\varphi\rangle$ under ρ, one must compute complete extender sets under ρ, and also under each possible extension to ρ

that extends the information about f and g. Thus, we must individually use each incremental extension in $\mathfrak{I}_t^s(\rho)$ (which includes ρ itself) as a starting point for computing extenders. When both s and t are constants, a complete extender set for $\langle s,t,\varphi\rangle$ is the set of all registries in $\mathfrak{I}_t^s(\rho)$ under which $s\ \varphi\ t$. When either s or t is not a constant, a complete extender set for $\langle s,t,\varphi\rangle$ is, by the definitions of $\mathfrak{O}_t^s(\varphi,\rho)$ and $\mathfrak{I}_t^s(\rho)$, the union of complete extender sets for all sets $\mathfrak{O}_t^s(\varphi,\rho_1)$ corresponding to each ρ_1 in $\mathfrak{I}_t^s(\rho)$. Thus, using the remark in the paragraph above, a complete extender set for $\langle s,t,\varphi\rangle$ is the set of all combined extenders for all orderals in all sets $\mathfrak{O}_t^s(\varphi,\rho_1)$ corresponding to each ρ_1 in $\mathfrak{I}_t^s(\rho)$. The minimal reduction of this set yields the minimal complete extender set for $\langle s,t,\varphi\rangle$.

Finally, $\mathfrak{E}(\succeq^{\rho})$, the minimal complete extender set for $s \succeq^{\rho} t$ under ρ, is the minimal complete extender set for the comparator $\langle s,t,\succeq^{\rho}\rangle$ under ρ, computed in the manner indicated above.

The function ComparatorExtenders, shown in Figure 3-2, computes and returns the minimal complete extender set for a given comparator under a given registry. The function OrderalExtenders, shown in Figure 3-3, computes and returns the set of all combined extenders for all orderals in a given set under a given registry. The two functions are mutually recursive.

ComparatorExtenders first accumulates, in S', a complete extender set for $\langle s,t,\varphi\rangle$ by collecting combined extenders, in the manner indicated above. The minimal reduction of S' is then accumulated in S to obtain the minimal complete extender set.

OrderalExtenders uses S to accumulate all combined extenders for all orderals, D, in S'. C holds the combined extenders for all comparators preceding $\langle s,t,\varphi\rangle$ in D. C' is used to incrementally accumulate the next value of C, as each minimal extender for $\langle s,t,\varphi\rangle$ is considered. OrderalExtenders assumes the existence of a subsidiary function, IsConsistent, that returns `true` if and only if its argument is a consistent registry.

Figure 3-4 presents the minimal complete extender set for an example comparator, as computed by the ComparatorExtenders function in Figure 3-2. The example is derived from one of the equations in Figure 3-1 on Page 47. Here, we assume that the current registry is empty.

Figure 3-2: Function to Compute the Minimal Extenders for a Comparator

```
function ComparatorExtenders ($\langle s,t,\varphi\rangle$, $\rho$) returns ($S$)

    Compute complete extender set:
    $S' :=$ {}
    if ($s$ is a constant) and ($t$ is a constant) then
        for each $\rho_1$ in $\mathcal{I}_t^s(\rho)$ do
            if $s\ \varphi\ t$ then $S' := S' \cup \{\rho_1\}$ endif
        endfor
    else  for each $\rho_1$ in $\mathcal{I}_t^s(\rho)$ do
            $S' := S' \cup$ OrderalExtenders($\mathfrak{I}_t^s(\varphi,\rho_1)$, $\rho_1$)
        endfor
    endif

    Compute minimal reduction of complete extender set:
    $S :=$ {}
    for each $\rho_1$ in $S'$ do
        for each $\rho_2$ in $S'$ do
            if $\rho_1$ is a strict extension of $\rho_2$ then $\rho_1 := \rho_2$ endif
        endfor
        $S := S \cup \{\rho_1\}$
    endfor
    return($S$)
end ComparatorExtenders
```

Figure 3-3: Function to Compute All Combined Extenders for All Orderals in a Set

```
function OrderalExtenders ($S'$, $\rho$) returns ($S$)
    $S$ := {}
    for each $D$ in $S'$ do

        Compute complete extender set for D:
        $C$ := {}
        for each $\langle s,t,\varphi \rangle$ in $D$ do
            $C'$ := {}

            Incrementally compute derived extenders:
            for each $\rho_1$ in ComparatorExtenders($\langle s,t,\varphi \rangle$, $\rho$) do
                for each $\rho_2$ in $C$ do
                    $\rho_3 := \rho_1 \cup \rho_2$
                    if IsConsistent($\rho_3$) then $C' := C' \cup \{\rho_3\}$ endif
                endfor
            endfor
            $C := C'$
        endfor
        $S := S \cup C$
    endfor
    return($S$)
end OrderalExtenders
```

Figure 3-4: Minimal Extenders for $\langle f(g(y, x), y), g(x, y), \succ^{\mathcal{P}}\rangle$ Under Empty Registry

(1) $\{\psi(g) = ●\}$

(2) $\{f \triangleright g\}$

(3) $\{f \trianglerighteq g, \psi(f) = ◐, \psi(g) = ◐\}$

(4) $\{f \trianglerighteq g, \psi(f) = ●, \psi(g) = ●\}$

(5) $\{f \trianglerighteq g, \psi(f) = ◐, \psi(g) = ●\}$

Lescanne has noted that computing $\mathfrak{S}(\succ^{\mathcal{D}})$ is roughly similar to computing $\mathfrak{S}(\succ^{\mathcal{P}})$. When comparing two terms, $\succ^{\mathcal{D}}$ first builds the *decompositions* of those terms, and compares these decompositions instead of comparing the terms directly. A *decomposition* is a multiset of *path decompositions*, a *path decomposition* is a multiset of *elementary decompositions*, and *elementary decompositions* consist of decompositions and path decompositions[11]. When computing minimal extenders for decompositions, one changes the definition of a *relator* to be $\succ^{\mathcal{D}}$, $\succeq^{\mathcal{D}}$, or $=^{\mathcal{D}}$, and a *comparator* becomes a pair of decompositions, together with a relator. The definition of *orderal* is changed accordingly, and the complete set of orderals is derived directly from the definition of the multiset ordering relational (Definition 7, Page 42). The new definition of *combined extender* follows from the new definition of orderal, and thus requires the ability to compute minimal extenders for path decompositions. The complete extender set and its minimal reduction are computed from the combined extenders in the same manner as for $\succ^{\mathcal{P}}$, to obtain the complete set of minimal extenders for ordering the two decompositions. Minimal complete extender sets are similarly obtained for path decompositions and elementary decompositions.

[11] We are glossing over many details here. When maximal choices are used for efficiency, decompositions and path decompositions are sets rather than multisets. Also, elementary decompositions may be *multiset tripartite*, *multiset quadripartite*, or *lexicographic*, depending on status and the [illegible].

3.5.2 Minimal Extenders for COS

This section presents the outline of a scheme to automatically generate the minimal complete extender sets under $\succ^{\underline{*}}$. This same technique can also be used to compute $\succ^{\underline{*}}$ itself. The scheme assumes the ability to compute minimal complete extender sets under $\succ^{\underline{P}}$ or $\succ^{\underline{D}}$, presented in the last section.

We do not propose that an implementation of $\mathfrak{E}(\succ^{\underline{*}})$ literally use the technique presented here. Our purpose is to demonstrate that $\mathfrak{E}(\succ^{\underline{*}})$ can be computed using implementations of $\mathfrak{E}(\succ^{\underline{P}})$ or $\mathfrak{E}(\succ^{\underline{D}})$, and that $\succ^{\underline{*}}$ need not be implemented by computing $\succ^{\underline{P}}$ under potentially thousands of registries. Further work is needed to discover an appropriate, practical implementation that might make use of the ideas presented in this section.

The minimal extenders for $\succ^{\underline{*}}$ are closely related to the minimal extenders for $\succ^{\underline{P}}$ and $\succ^{\underline{D}}$. By the definition of $\succ^{\underline{*}}$, the set of all total extensions to the registries in $\mathfrak{E}(\succ^{\underline{*}})$ is the set of all total registries under which $s \succ^{\underline{P}} t$. The same is true for $\mathfrak{E}(\succ^{\underline{P}})$. The difference between $\mathfrak{E}(\succ^{\underline{*}})$ and $\mathfrak{E}(\succ^{\underline{P}})$ is that the extensions in $\mathfrak{E}(\succ^{\underline{P}})$ are not necessarily minimal for $\succ^{\underline{*}}$. Thus, we propose that $\mathfrak{E}(\succ^{\underline{*}})$ be obtained by properly reducing the registries in $\mathfrak{E}(\succ^{\underline{P}})$. Since $\succ^{\underline{P}}$ and $\succ^{\underline{D}}$ are the same ordering under total registries, the same relationship holds between $\mathfrak{E}(\succ^{\underline{*}})$ and $\mathfrak{E}(\succ^{\underline{D}})$ as between $\mathfrak{E}(\succ^{\underline{*}})$ and $\mathfrak{E}(\succ^{\underline{P}})$. For concreteness, we will use $\mathfrak{E}(\succ^{\underline{P}})$ here, though $\mathfrak{E}(\succ^{\underline{D}})$ could be used in exactly the same manner.

Our approach to computing $\mathfrak{E}(\succ^{\underline{*}})$ involves viewing registries as formulas in propositional calculus. Every registry can be viewed as a set of *items*, where an *item* is a stated $\unrhd$ or $\not\trianglelefteq$ relationship between two operators, or a status assignment to some operator. For example, $f \unrhd g$, $f \not\trianglelefteq g$, and $\psi(f) =$ Ⓜ are three items. (All $\rhd$ shorthands are represented by their two-item $\unrhd$ and $\not\trianglelefteq$ equivalents, and $\doteq$ is represented by two $\unrhd$ items.) For any registry, ρ, we define its propositional formula, denoted Prop(ρ), to be the Boolean conjunction of all items comprising ρ. Thus, if ρ is:

$$\{f \unrhd g, g \unrhd h, f \unrhd h, \psi(f) = \text{Ⓜ}\}$$

Prop(ρ) is:

$$f \unrhd g \wedge g \unrhd h \wedge f \unrhd h \wedge \psi(f) = \text{Ⓜ}$$

Note that for two registries, ρ_1 and ρ_2, ρ_1 is an extension of ρ_2 if and only if Prop(ρ_1) $\Rightarrow$ Prop(ρ_2).

An extender set can be viewed as a formula in disjunctive normal form, by taking the disjunction of the formulas associated with each of the extenders in the set. For example, the extender set in Figure 3-4 can be viewed as the formula shown in Figure 3-5. We use Prop(S) to denote the formula associated with the extender set S. Note that ρ_1 is an extender for $s \succ t$ under ρ if and only if Prop(ρ_1) $\Rightarrow$ Prop($\mathfrak{X}^S_t(\succ,\rho)$).

Figure 3-5: Formula Formed from the Extenders in Figure 3-4

$$
\begin{array}{l}
(\psi(g) = \textcircled{M}) \vee \\
[(f \dot{\geq} g) \wedge (f \dot{\not\geq} g)] \vee \\
[(f \dot{\geq} g) \wedge (\psi(f) = \textcircled{L}) \wedge (\psi(g) = \textcircled{L})] \vee \\
[(f \dot{\geq} g) \wedge (\psi(f) = \textcircled{R}) \wedge (\psi(g) = \textcircled{R})] \vee \\
[(f \dot{\geq} g) \wedge (\psi(f) = \textcircled{L}) \wedge (\psi(g) = \textcircled{R})]
\end{array}
$$

A formula in disjunctive normal form that is composed of items, where none of the items is negated, can be straightforwardly viewed as a set of registries, provided each disjunct forms a consistent registry. We use Reg(η) to denote the set of registries obtained from such a formula, η. By taking these two different views of an extender set, one can manipulate the extenders in the well-understood domain of propositional calculus, but interpret the formulas in the domain of registry extensions. We propose a method for computing $\succ^{*}$ extenders that is based on propositional calculus manipulation of $\succ^{P}$ extenders.

Let S be the set of operators appearing in the rewriting system of interest, and T be the set of operators appearing in s and/or t. Note that for any S, there exists a formula, Consis(S), such that a registry, ρ, over S is consistent if and only if Prop(ρ) $\wedge$ Consis(S) is **true**. (The formula Consis(S) is easily constructed from S using the definition of consistent registry.) Also note that $\mathcal{A}^S_t(\{\})$ is the complete set of all registries that are total over T. For any $\rho_2 \in \mathcal{A}^S_t(\{\})$, ρ_2 is a total extension, over T, of a registry ρ_1, over S, if and only if $\rho_1 \cup \rho_2$ is a consistent registry; or, in terms of formulas, if and only if Prop(ρ_1) $\wedge$ Prop(ρ_2) $\wedge$ Consis(S) is **true**.

By the definition of $\succ^{*}$, ρ_1 is an extender for $s \succ^{*}[\rho]\ t$ if and only if all total extensions of ρ_1 over T are extenders for $s \succ^{P}[\rho]\ t$. Translating to formulas, ρ_1 is an extender for $s \succ^{*}[\rho]\ t$ if and only if

$(\forall \rho_2 \in \mathcal{A}^s_t(\{\}))\ [(\mathrm{Prop}(\rho_1) \wedge \mathrm{Prop}(\rho_2) \wedge \mathrm{Consis}(S)) \Rightarrow \mathrm{Prop}(\mathfrak{E}^s_t(\succ^{P},\rho))]$

The above formula may be syntactically transformed into

$$\mathrm{Prop}(\rho_1) \Rightarrow [\mathrm{Prop}(\mathfrak{E}^s_t(\succ^{P},\rho)) \vee \neg\mathrm{Prop}(\mathcal{A}^s_t(\{\})) \vee \neg\mathrm{Consis}(S)]$$

Therefore, ρ_1 is an extender for $s \succ^{\star} t$ if and only if the above implication holds. Let η denote a disjunctive normal form of $\mathrm{Prop}(\mathfrak{E}^s_t(\succ^{P},\rho)) \vee \neg\mathrm{Prop}(\mathcal{A}^s_t(\{\})) \vee \neg\mathrm{Consis}(S)$, where each disjunct contains as few items (or negated items) as possible under the simplification rules of propositional calculus. Since $\mathrm{Prop}(\rho_1)$ is a conjunction of items, $\mathrm{Prop}(\rho_1) \Rightarrow \eta$ if and only if $\mathrm{Prop}(\rho_1)$ implies one of the disjuncts in η. By its construction, $\mathrm{Prop}(\rho_1)$ contains no negated items, so all disjuncts in η that contain any negated items may be removed from η without affecting the Boolean implication. Let $\mathrm{Reduce}(s,t,S,\rho)$ denote this reduced form of η.

We now have that ρ_1 is an extender for $s \succ^{\star} t$ if and only if

$$\mathrm{Prop}(\rho_1) \Rightarrow \mathrm{Reduce}(s,t,S,\rho)$$

Interpreting this in the domain of extenders, $\mathrm{Reg}(\mathrm{Reduce}(s,t,S,\rho))$ is a complete extender set for $s \succ^{\star} t$ under ρ. The extenders in $\mathrm{Reg}(\mathrm{Reduce}(s,t,S,\rho))$ are already minimal, because the disjuncts in $\mathrm{Reduce}(s,t,S,\rho)$ contain as few items as possible, so $\mathrm{Reg}(\mathrm{Reduce}(s,t,S,\rho))$ is $\mathfrak{E}(\succ^{\star})$.

In short, we may compute $\mathfrak{E}(\succ^{\star})$ by first computing $\mathfrak{E}(\succ^{P})$, and then manipulating $\mathfrak{E}(\succ^{P})$ using propositional calculus. Furthermore, we have $s \succ^{\star}[\rho]\ t$ if and only if the computed value of $\mathfrak{E}(\succ^{\star})$ is $\{\rho\}$. This gives us an alternative method for computing the $\succ^{\star}$ registered ordering itself, without having to compare s and t with $\succ^{P}$ under many registries.

As an example of computing $\succ^{\star}$ extenders, consider the terms $s = f(f(a, a), f(b, b))$ and $t = g(b, a)$. The extenders comprising $\mathfrak{E}^s_t(\succ^{P},\rho)$, assuming that ρ is empty, are shown in Figure 3-6. Note that $f \rhd g$ is a minimal extender for $s \succ^{P} t$, as indicated in the figure, but $f \unrhd g$ is not an extender for $s \succ^{P} t$. However, the formula $\mathrm{Reduce}(s,t,S,\rho)$ for this example is the single item $f \unrhd g$. Thus, this yields the only minimal extender for $s \succ^{\star} t$: $\mathfrak{E}(\succ^{\star}) = \{f \unrhd g\}$.

Figure 3-6: Minimal Extenders for $f(f(a, a), f(b, b)) \succ^{P} g(b, a)$ Under Empty Registry

(1) $\{f > g\}$

(2) $\{f \geq g, \psi(f) = Ⓜ, \psi(g) = Ⓜ\}$

(3) $\{f \geq g, \psi(f) = Ⓛ, \psi(g) = Ⓡ\}$

(4) $\{f \geq g, \psi(f) = Ⓡ, \psi(g) = Ⓛ\}$

(5) $\{f \geq g, f \geq b, \psi(f) = Ⓛ, \psi(g) = Ⓛ\}$

(6) $\{f \geq g, a \geq b, \psi(f) = Ⓛ, \psi(g) = Ⓛ\}$

(7) $\{f \geq g, b \geq a, \psi(f) = Ⓛ, \psi(g) = Ⓛ\}$

(8) $\{f \geq g, f \geq a, \psi(f) = Ⓡ, \psi(g) = Ⓡ\}$

(9) $\{f \geq g, a \geq b, \psi(f) = Ⓡ, \psi(g) = Ⓡ\}$

(10) $\{f \geq g, b \geq a, \psi(f) = Ⓡ, \psi(g) = Ⓡ\}$

3.6 Automatically Constructing Rewriting Systems

With an automatic ordering, $\succ$, a terminating rewriting system can be automatically constructed from a set of equations, $\mathcal{E}$, as follows: Start with an empty registry. Consider each equation, $s = t$, in $\mathcal{E}$. If there are any minimal extenders for $s = t$ in either direction, choose one of them to be the current registry, and go on to the next equation. Otherwise, back up to the last equation, choose one of its minimal extenders that has not yet been tried, and continue. When $s = t$ is considered again, the registry might be such that it has some minimal extenders. Systematically pursued, this automatic technique is a depth-first search for a terminating registry for $\mathcal{E}$ under $\succ$. If all minimal extenders are tried at each backtrack point, and the depth-first search fails to find such a registry, there is no terminating registry. In this case, either the equations in $\mathcal{E}$ cannot be ordered into a terminating rewriting system, $\mathcal{R}$, or $\succ$ is not powerful enough to demonstrate the termination of $\mathcal{R}$.

Figure 3-7 presents a procedure, AutomaticConstruction, that formalizes the above idea. This procedure has been implemented in REVE. AutomaticConstruction takes $\mathcal{E}$, and returns a terminating registry for $\mathcal{E}$ and the automatic ordering $\succ$, together with the terminating rewriting system corresponding to that registry. If there exists no terminating registry for $\mathcal{E}$ under $\succ$, AutomaticConstruction halts with "failure." The procedure makes use of the stack primitives New, Push, Pop, Top, and IsEmpty, which have their conventional meanings. The AnyOf function returns any element of its set argument, and EmptyRegistry returns an empty registry. As each equation is considered, it is removed from $\mathcal{E}$. When an equation is successfully ordered, a tuple consisting of the following items is pushed onto the stack:

- The equation, in the direction it is being considered.
- A Boolean value that indicates whether the equation has been tried in the reverse direction.
- The minimal extenders, for this direction of the equation, that have not yet been tried.
- The rest of $\mathcal{E}$.

Whenever there are no extenders, in either direction, for some equation, the stack is popped until an equation is found for which there are minimal extenders that have not yet been tried, and the current contents of $\mathcal{E}$ are reset accordingly. If all equations are successfully pushed onto the stack, the current registry is returned, along with a rewriting system consisting of all the equations in the direction that they appear on the stack. Note that $\mathfrak{X}_t^s(\succ,\rho)$ consists only of ρ if $s \succ t$ under ρ. Also, if $s \rightarrow t$ is not a valid rewrite rule, we have $s \not\succ t$ (because $\succ[\rho]$ is a simplification ordering), so $\mathfrak{X}_t^s(\succ,\rho) = \{\}$ in this case.

Figure 3-7: Procedure To Automatically Construct a Terminating Rewriting System

```
procedure AutomaticConstruction ($\mathcal{E}$) returns ($\rho, \mathcal{R}$)
    stack := New
    $\rho$ := EmptyRegistry
    while $\mathcal{E} \neq \{\}$ do
        $s = t$ := AnyOf($\mathcal{E}$); $\mathcal{E} := \mathcal{E} - \{s = t\}$
        reversed := false
        $X := \mathfrak{S}_t^s(\succ, \rho)$
        while $X = \{\}$ do
            if $\neg$reversed then
                $s, t := t, s$
                reversed := true
                $X := \mathfrak{S}_t^s(\succ, \rho)$
            else  if IsEmpty(stack) then halt with failure endif
                $\langle s = t$, reversed, $X, \mathcal{E} \rangle$ := Top(stack)
                stack := Pop(stack)
            endif
        endwhile
        $\rho$ := AnyOf($X$); $X := X - \{\rho\}$
        stack := Push(stack, $\langle s = t$, reversed, $X, \mathcal{E} \rangle$)
    endwhile
    $\mathcal{R} := \{\}$
    while $\neg$IsEmpty(stack) do
        $\langle s = t$, reversed, $X, \mathcal{E} \rangle$ := Top(stack)
        stack := Pop(stack)
        $\mathcal{R} := \mathcal{R} \cup \{s \rightarrow t\}$
    endwhile
    return($\rho, \mathcal{R}$)
end AutomaticConstruction
```

Figure 3-8: Example to Illustrate the AutomaticConstruction Procedure

(1) $f(f(x, x), y) = f(x, f(x, y))$

(2) $f(g(y, x), y) = g(x, y)$

(3) $g(f(y, x), x) = f(x, y)$

(4) $g(g(x, x), y) = g(y, g(x, y))$

Consider the operation of AutomaticConstruction on the set of equations in Figure 3-8, which is the same set as in Figure 3-1 on Page 47. We will use $\succ^{\mathcal{L}}$ as the automatic ordering, and consider the equations in the order shown in the figure.

(1) The current registry is empty. The first equation is not orderable under an empty registry, so we compute $\mathfrak{E}(\succ^{\mathcal{L}})$, which contains the single extender $\{\psi(f) = \ominus\}$. Following AutomaticConstruction, we use this extender as the new current registry, and push, onto the stack, a tuple consisting of the first equation (in the direction shown in Figure 3-8), false, an empty extender, and the other three equations.

(2) The second equation is not orderable under the current registry. For this equation, $\mathfrak{E}(\succ^{\mathcal{L}}) = \{\psi(f) = \ominus, \psi(g) = \oplus\}$ and $\{\psi(f) = \ominus, f \succeq g\}$. We arbitrarily choose the first extender as the new current registry, and push, onto the stack, a tuple consisting of the second equation, false, the extender set $\{\{\psi(f) = \ominus, f \succeq g\}\}$, and the last two equations.

(3) The third equation is not orderable under the current registry. Here, $\mathfrak{E}(\succ^{\mathcal{L}})$ is the singleton set $\{\psi(f) = \ominus, \psi(g) = \oplus, g \succ f\}$. We make this extender the current registry, and push a tuple whose extender set is empty onto the stack.

(4) The fourth equation is not orderable under the current registry, and there are no minimal extenders in either direction. Hence, we return to the third equation and pop the stack, as per AutomaticConstruction.

(5) There are no further extenders in either direction for the third equation, so we return to the second equation and pop the stack.

(6) There is one remaining minimal extender, in the left-to-right direction, for the second equation, namely $\{\psi(f) = \ominus, f \succeq g\}$. We use this as the new current registry, and push a tuple whose extender set is empty.

(7) AutomaticConstruction now reconsiders the third equation. It is not orderable

under the current registry, but $\mathfrak{E}(\succeq^{P})$ consists of a single minimal extender (and it is different from Step (3), above): $\{\psi(f) = Ⓛ, f \doteq g\}$. We choose this as the current registry, and push a tuple containing an empty extender set onto the stack.

(8) The fourth equation is not orderable under the current registry, but this time the minimal complete extender set in the reverse direction is $\{\psi(f) = Ⓛ, f \doteq g, \psi(g) = Ⓡ\}$. We use this as the current registry, and push a tuple on the stack containing the reversed equation.

(9) There are no further equations to consider, so we pop the equations from the stack, build them into a rewriting system, and return the current registry together with that rewriting system.

The registry $\{\psi(f) = Ⓡ\}$ orders the first equation in Figure 3-8 in the reverse direction. If this equation were reversed in the figure, AutomaticConstruction would try several registry extensions before finally backing up to the first equation, reversing it, and continuing.

As an aside, it is not strictly necessary to use only minimal extenders when proving termination automatically with registered orderings. Allowing non-minimal extenders can sometimes lead to a gain in efficiency. For example, consider the minimal extenders in Figure 3-4 on Page 59. The second minimal extender in the figure states that all registries that contain $f \gtrdot g$ are extenders for $s \succeq^{P} t$. If AutomaticConstruction cannot finish successfully using extensions of this extender, it makes no sense to try the third, fourth, and fifth minimal extenders in conjunction with $f \not\geq g$. Thus, one may replace all occurrences of $f \geq g$ in the figure with $f \doteq g$, without danger of AutomaticConstruction missing a potential extender. Making this replacement may allow the rewriting system construction process to proceed faster, since then the extensions of the second extender will be disjoint from the extensions of the last three extenders, avoiding some potential redundancy when searching for an extender for the rewriting system. Once $\mathfrak{E}(\succ)$ has been computed, one may perform a postprocessing on $\mathfrak{E}(\succ)$ to remove such redundancies before considering the extender set in AutomaticConstruction, if desired.

Instead of arbitrarily choosing a minimal extender from $\mathfrak{E}(\succ)$, an implementation of AutomaticConstruction might display the extenders in $\mathfrak{E}(\succ)$ and permit the user to select one. Rather than presenting the entirety of each extender to the user, it may be desirable to present the *transitive reduction* of each extender, for brevity. A *transitive reduction* [Aho 72] of a directed graph, G_1, is a smallest graph, G_2, such that the transitive closures of G_1 and G_2

are the same. The relation $\underline{\gtrdot}$ in a precedence can be regarded as a directed graph, where operators are nodes, and $\underline{\gtrdot}$ defines the edges on those nodes. We define the *transitive reduction* of a registry, $\langle\langle\underline{\gtrdot},\not\lessgtr\rangle,\psi\rangle$, to be the transitive reduction of $\underline{\gtrdot}$, together with $\not\lessgtr$ and ψ, which remain unchanged. The transitive reduction of ρ conveys the same information as ρ. It may also be desirable to subtract away the current registry before presenting the transitive reduction of each extender, so that only the new information introduced by the extender is displayed.

One might think that AutomaticConstruction's exhaustive backtracking scheme for constructing a terminating rewriting system would be too slow to be practical. However, we have found that for typical examples where termination can be proven using $\succ$, backtracking is usually not required. Even though there may be many extenders to choose from when an equation is unorderable, successive extender choices have a cumulative effect such that the terminating registry obtained tends to be relatively insensitive to the particular extender choices made along the way.

3.7 Summary

In this chapter, we have presented the basic definitions of relations and orderings, and introduced *relationals* as parameterized relations. We then presented simplification orderings, and the termination theorem that justifies the use of simplification orderings in termination proofs. The notion of a *registered ordering* was defined: a relational, parametized on a registry, that yields a stable simplification ordering. We then introduced *automatic orderings*, which are registered orderings whose implementation can compute the *minimal complete extender set* when two terms are unorderable.

We described RPOS, which can be viewed as a registered ordering, and extended it into EPOS, which is more suitable for the automatic construction of terminating rewriting systems. This was followed by a brief discussion of RDOS, the important role that RDOS has played in establishing the utility and viability of helping the user dynamically extend the registry when two terms are unorderable, and the fact that RDOS can be extended slightly to produce EDOS. We then presented COS, which is more powerful than EPOS and EDOS, and proved the correctness of COS in the context of termination proofs.

This was followed by algorithms that allow EPOS, EDOS, and COS to be used as automatic orderings. A minimal complete extender set scheme for EPOS was described in detail, and we roughly indicated how the scheme could be modified for EDOS. We showed how, in principle, the minimal extenders under COS, and the COS registered ordering itself, could be computed using minimal extender schemes for either EPOS or EDOS.

Finally, we presented a procedure that automatically constructs a terminating rewriting system from a set of equations. The procedure makes use of an automatic ordering, and automatic implementations of EPOS, EDOS, or COS could be used for this purpose.

Chapter Four

A Failure-Resistant Knuth-Bendix Design

4.1 Introduction

In its original formulation (Section 2.6), the Knuth-Bendix completion procedure is used to transform a term rewriting system, $\mathfrak{R}$, into another rewriting system, $\mathfrak{R}'$, such that $\mathfrak{R}'$ is convergent and $=_{\mathfrak{R}}$ equals $=_{\mathfrak{R}'}$. As discussed in Section 2.5, $\mathfrak{R}'$ provides a decision procedure for $=_{\mathfrak{R}}$. However, Knuth-Bendix is not an algorithm: it may halt with "failure" if the two sides of a rule are not orderable, or fail to terminate because it may generate an infinite set of rules.

The original version of Knuth-Bendix, as presented in Figures 2-6 and 2-7 on Pages 28 and 29, was chosen by its authors for its simplicity of exposition and for ease of proving its correctness, rather than for its efficiency. It differs slightly from later formulations by others in that it begins with a set of previously-ordered rewrite rules to be completed, rather than starting with a set of equations and using the reduction ordering to orient each of those equations into a rewrite rule. Three important problems of the original procedure are:

(1) It is inefficient,

(2) It fails whenever an unorderable equation is generated, and

(3) The reduction ordering must be given *a priori*.

This chapter presents a new, failure-resistant formulation of Knuth-Bendix that addresses these issues. As a partial solution to (1), above, it incorporates improved schemes for generating critical pairs and normalizing the rewriting system. For (2), it uses a fine-grained approach to postponing equations that are currently unorderable. For (3), it makes use of an important idea that first appeared in Lescanne's REVE 1: it allows the ordering to be incrementally extended as unorderable equations are encountered. The net result is a potentially faster completion procedure that halts with "failure" in fewer cases. The procedure is formulated as a sequence of tasks, that are performed in an order commensurate with their expected contribution to the successful and expeditious completion of the procedure.

Existing "incremental" implementations of orderings provide various degrees of help to the user when an unorderable equation is encountered. At one end of the user-assistance spectrum, automatic orderings compute all the possible ways that the registry can be extended to allow the equation to be ordered. At the other end, if no ordering implementation at all is used, the user must hand-order each equation with no assistance from the program. In between are registered orderings, such as EDOS, where current implementations provide suggestions that help the user find an appropriate registry extension[12]. We assume here that ordering "extensions" do not change the ordering of previously-ordered rules in the rewriting system. (This is true of all registered orderings described in Chapter 3, because they are monotonic in the registry.)

If partial help or no help is provided to the user to extend the ordering, discovering appropriate ordering extensions can be a slow process for the user, so Knuth-Bendix can usually be expedited in this case by postponing unorderable equations for a time. This may allow generated critical pairs to become rewrite rules that reduce some of these unorderable equations, to make the equations orderable or make them disappear.

If an automatic ordering is used, it is usually faster to compute the minimal complete extender set for unorderable equations before generating more critical pairs. This is because Knuth-Bendix typically generates the smallest, most useful equations first, and, with automatic orderings, the overhead of searching for an appropriate registry extension is reduced. In this case, an equation should probably only be postponed if there exists no registry under which it is ordered.

Both of these possibilities are considered here. Section 4.2 describes Huet's improved version of Knuth-Bendix. Section 4.3 presents our standard failure-resistant Knuth-Bendix scheme, especially appropriate for implementations of orderings that provide only partial help for extending the ordering. Section 4.4 indicates how the procedure can be appropriately modified for automatic orderings by switching two of the Knuth-Bendix tasks. Both of the failure-resistant schemes are provided in REVE.

[12]As noted in Section 3.5, Lescanne is currently working on an automatic ordering implementation of EDOS.

4.2 Huet's Version

Huet's formulation of Knuth-Bendix [Huet 81] is presented in Figure 4-2, which makes use of the functions in Figure 4-1. The initial input to the procedure is a reduction ordering, $\succ$, and a set of equations, $\mathcal{E}$. The rewriting system, $\mathcal{R}$, is represented by a set of triples. Each triple consists of a rewrite rule, an integer label, and a flag, in that order. If the flag is $\bullet$, the rewrite rule is considered to be "marked"; if the flag is $\circ$, the rule is "unmarked." Since the procedure preserves the invariant that no rewrite rule occurs in more than one triple in $\mathcal{R}$, a triple can be denoted by its rewrite rule. As in Figure 2-7 on Page 29, **repeat** means "go to the first statement of the smallest enclosing loop."

Figure 4-1: Auxiliary Functions Used by Figure 4-2

Normal(t, $\mathcal{R}$)	$\equiv$ A normal form of the term t with respect to the rewriting system $\mathcal{R}$
Unorderable($s = t$)	$\equiv$ ($s \not\succ t$) **and** ($t \not\succ s$)
Order($s = t$)	$\equiv$ **if** $s \succ t$ **then** $s \rightarrow t$ **else** $t \rightarrow s$
CriticalPairs(r, r')	$\equiv$ Set of all critical pairs between the rules r and r'
AnyOf($\mathcal{E}$)	$\equiv$ Any equation in the set $\mathcal{E}$

Huet's version of Knuth-Bendix is more efficient than the original. It achieves this efficiency with two key optimizations:

- Huet's procedure generates the critical pairs between any two rewrite rules only once, whereas the original procedure begins again to look for critical pairs among all rules during each iteration through the main loop. The speed-up attained in Huet's formulation can be substantial, since the unifications required in computing critical pairs can be time consuming.
- Huet's procedure does not "normalize" the entire rewriting system each time a rewrite rule is added. Rather, it uses the fact that the rewriting system is completely normalized prior to adding an additional rewrite rule, and that only those rewrite rules whose left or right-hand sides can be rewritten by the new rule will not be in normal form once the rule has been added. Furthermore, unlike the original Knuth-Bendix, Huet's procedure does not re-order rewrite rules whose right-hand sides are rewritten during normalization but whose left-hand sides are left intact. This re-ordering is unnecessary because such rules will still be ordered under the reduction ordering.

Figure 4-2: Huet's Formulation of the Knuth-Bendix Completion Procedure

```
$\mathfrak{R} := \{\}; n := 0$
loop
    while $\mathcal{E} \neq \{\}$ do

        Find non-joinable critical pair:
        $(s = t) :=$ AnyOf($\mathcal{E}$)
        $\mathcal{E} := \mathcal{E} - \{s = t\}$
        $s' :=$ Normal($s, \mathfrak{R}$); $t' :=$ Normal($t, \mathfrak{R}$)
        if $s' = t'$ then repeat endif

        Order equation:
        if Unorderable($s' = t'$) then halt with failure endif
        $(\lambda \to \rho) :=$ Order($s' = t'$)

        Normalize rewriting system:
        for each $\langle \gamma \to \mu, i, \Omega \rangle$ in $\mathfrak{R}$ do
            $\gamma' :=$ Normal($\gamma, \{\lambda \to \rho\}$)
            if $\gamma \neq \gamma'$ then
                $\mathfrak{R} := \mathfrak{R} - \{\gamma \to \mu\}$; $\mathcal{E} := \mathcal{E} \cup \{\gamma' = \mu\}$
            else  $\mu' :=$ Normal($\mu, \mathfrak{R} \cup \{\lambda \to \rho\}$)
                if $\mu \neq \mu'$ then $\mathfrak{R} := (\mathfrak{R} - \{\gamma \to \mu\}) \cup \{\langle \gamma \to \mu', i, \Omega \rangle\}$ endif
            endif
        endfor

        $n := n + 1$
        $\mathfrak{R} := \mathfrak{R} \cup \{\langle \lambda \to \rho, n, \circ \rangle\}$
    endwhile

    Find an unmarked rule:
    for each $\langle \lambda \to \rho, i, \Omega \rangle$ in $\mathfrak{R}$ do
        if $\Omega = \circ$ then goto Compute critical pairs endif
    endfor
    halt with success

    Compute critical pairs:
    for each $\langle \gamma \to \mu, k, \Phi \rangle$ in $\mathfrak{R}$ do
        if $k \leq i$ then $\mathcal{E} := \mathcal{E} \cup$ CriticalPairs($\lambda \to \rho, \gamma \to \mu$) endif
    endfor

    Mark the rule:
    $\mathfrak{R} := (\mathfrak{R} - \{\langle \lambda \to \rho, i, \circ \rangle\}) \cup \{\langle \lambda \to \rho, i, \bullet \rangle\}$
endloop.
```

Huet's Knuth-Bendix implementation removes some of the obvious inefficiencies of the original procedure. However, it is still the case that the reduction ordering must be given *a priori*, and it fails whenever an unorderable equation is generated. The next section describes how REVE's Knuth-Bendix implementation attempts to address the latter two problems.

4.3 A Failure-Resistant Knuth-Bendix

This section presents REVE's failure-resistant Knuth-Bendix implementation. ([Forgaard 84] presents a preliminary version of these results[13].) The chief improvements of this version over Huet's are:

- REVE does not require that the reduction ordering be given *a priori*. The ordering may be extended during the course of running Knuth-Bendix. During this process, the user may undo previous decisions and restart Knuth-Bendix.
- REVE's Knuth-Bendix implementation does not fail when an unorderable equation is found. The ordering may be extended to allow the equation to be ordered, or the equation may be postponed. Postponement might allow the equation to be reduced, to disappear, or to be ordered later.
- REVE automatically postpones consideration of large equations.
- REVE computes smaller critical pairs first, which can expedite the completion process.
- REVE's Knuth-Bendix incorporates the modification shown in Figure 2-9 on Page 33, to support the Huet-Hullot inductionless induction method (see Section 2.7).

REVE's technique of computing small critical pairs is presented in Section 4.3.1. Section 4.3.2 describes the use of user interaction in extending the ordering. Section 4.3.3 describes equation postponement in REVE, and Section 4.3.4 outlines REVE's scheme for efficiently computing the normal forms of postponed equations. Finally, the task-based control flow in REVE's Knuth-Bendix implementation is presented in Section 4.3.5.

[13]The *flexible* attribute, described in [Forgaard 84], is unnecessary here, because we assume monotonicity in the ordering.

4.3.1 Computing Small Critical Pairs

Huet's scheme for computing critical pairs can be characterized as follows: Maintain the rewriting system as a list of rules. Each rule that gets added to the list is initially unmarked. In the critical pair computation step, choose an unmarked rule $\lambda \rightarrow \rho$ and compute all critical pairs between $\lambda \rightarrow \rho$ and itself, and between $\lambda \rightarrow \rho$ and every rule above it in the list. Then, mark $\lambda \rightarrow \rho$. In this way, each distinct pair of rewrite rules is used only once.

In [Knuth 70], the authors note that small pairs of rewrite rules are more likely to lead to small critical pairs. Small critical pairs are useful because they take less time to generate and tend to lead to more general rules than do larger critical pairs. It is often the case that these rules reduce larger rules and equations, thus reducing the number of larger critical pairs that need to be generated.

Huet's Knuth-Bendix will tend to generate small critical pairs if it chooses the smallest unmarked rewrite rule when computing critical pairs, thus using unmarked rules in increasing order of size.[14] However, this refinement does not always pick the smallest *pair* of rules that has yet to be considered, since there may be rules in the list, below the chosen rule $\lambda \rightarrow \rho$, that are smaller than some of the rules above $\lambda \rightarrow \rho$ in the list. This strategy will tend to generate smaller critical pairs before larger ones, so the latter problem can be partially alleviated by always appending new rules to the bottom of the list so that larger rules tend toward the bottom.

If the list of rewrite rules is always maintained so that it is sorted by size, and if critical pairs with a chosen rule $\lambda \rightarrow \rho$ are calculated with rules above $\lambda \rightarrow \rho$ in order from the top of the list down to $\lambda \rightarrow \rho$ itself, the marking scheme will ensure that critical pairs are always calculated starting with smallest pair of rules that have not yet been considered.

REVE uses a strategy for choosing pairs of rules that is a compromise between the above two schemes. The REVE method does not pair the chosen rule with large unmarked rules, nor does it incur the additional mechanism (and minor inefficiencies) associated with maintaining a sorted list of rewrite rules. Instead, REVE chooses the smallest unmarked rule $\lambda \rightarrow \rho$, marks it, and then computes critical pairs between $\lambda \rightarrow \rho$ and every marked rule, including itself.

[14]This was the scheme employed in REVE 1.

Since all marked rules are "small" in the sense that each marked rule was at one time the smallest unmarked rule, this scheme tends to start with small pairs of rules and move up to the larger rules as Knuth-Bendix progresses. However, REVE's method does not necessarily start with the smallest pair of rules that have not yet been used, since some critical pairs may get generated that are smaller than some of the marked rules. Note that REVE's strategy, as with Huet's, considers each possible pair of rules, and does so only once.

Further ideas for computing small critical pairs first are presented in Section 6.2.4.1.

4.3.2 Proving Termination Using User Interaction

REVE's Knuth-Bendix provides explicit support for orderings that give help to the user when an equation is unorderable, although any ordering (including ordering by hand) may be used. The ordering is chosen by the user. We assume here that some ordering that provides help has been selected.

When REVE encounters an equation that the ordering is currently unable to order, the equation is shown to the user. He is also presented with any suggestions provided by the ordering. The user is then asked to choose an action from an appropriate subset of the choices shown in Figure 4-3.

If the user picks Choice (1), the ordering is extended accordingly and the equation becomes ordered into a rewrite rule in the appropriate direction.

Choice (2) puts the equation on the list of *unoriented* or *incompatible* equations. Choice (3) puts the equation on the *deferred* list. See the next section for a discussion of these lists.

If the user selects Choices (4) or (5), the equation gets added to the rewriting system, and REVE proceeds to try to complete it. (These choices are only allowed if the equation can be validly viewed as a rewrite rule in the selected direction.) If REVE succeeds, the user is warned that the resulting rewriting system may not be convergent because it is not guaranteed to terminate. Allowing this hand-ordering of equations is sometimes useful with equations that are not amenable to termination proof using the selected ordering.

Choice (6) invokes the technique, discussed in Section 2.4, for converting one equation into two. The user is prompted to supply the new operator name.

Figure 4-3: Choices for Unorderable Equations

(1) Extend the ordering in some manner, presumably using the suggestions provided by the ordering's implementation.

(2) Postpone the equation for the time being.

(3) Defer the equation.

(4) Accept the equation as a rewrite rule in the direction shown.

(5) Accept the equation as a rewrite rule in the reverse direction.

(6) Divide the equation into two equations, introducing a new operator.

(7) Interrupt Knuth-Bendix.

(8) Undo Knuth-Bendix.

If the user chooses Choice (7), REVE remembers the current state of Knuth-Bendix so that it can later be resumed from this point, and returns to the top command level.

Choice (8) causes REVE to restore the state of Knuth-Bendix to the most recent user decision point. The user may "undo" multiple times to return to any earlier interaction, and resume Knuth-Bendix from that point.

4.3.3 Postponing Equations

Both the original presentation of Knuth-Bendix and Huet's version halt with "failure" as soon as an equation or rewrite rule is found that cannot be ordered using the initial reduction ordering. However, as noted in [Dershowitz 85b], one can postpone consideration of unorderable equations, and abort only if there are no orderable ones. There are several good reasons for doing this. An unorderable equation might later become orderable, or disappear entirely, when it is reduced by a new rewrite rule. It might later become orderable when the ordering has been extended. Alternatively, the user may decide later to hand-order it (if it can be viewed as a rewrite rule), or to divide it into two equations.

In Section 4.3.1, we mentioned the usefulness of generating small critical pairs first. For similar reasons, it is advantageous to consider small equations first. Thus, REVE postpones large equations, in addition to the unorderable ones.

REVE's Knuth-Bendix implementation partitions equations into five lists. The equations that REVE has not yet tried to order are in the *new* list. The postponed unorderable equations are in one of the *incompatible*, *unoriented*, or *deferred* lists. The postponed large equations are in the *big* list.

An *incompatible* equation is one that cannot be viewed as a rewrite rule in either direction, as discussed in Section 2.5.

An *unoriented* equation is one that can be viewed as a rewrite rule, but is unorderable at the present time.

A *deferred* equation is an incompatible equation, or an unoriented equation that the user believes will probably never be orderable. It is being postponed, rather than divided or hand-ordered, because the user hopes that a later rule will reduce the equation to make it orderable. For example, the user should direct REVE to put cyclic equations, e.g., $x + y = y + x$, on the deferred list. In the future, REVE could be made to automatically put certain types of equations on the deferred list.

REVE's Knuth-Bendix implementation does not look at big equations until all other equations have been ordered or postponed, and all critical pairs have been computed. The number of symbols in every big equation is greater than or equal to β, a special value maintained by REVE. The size of all other postponed equations is less than β. The value of β is set so that no user-introduced equation is considered big. When REVE finally looks at the big equations, it considers them from smallest to largest, and the value of β is adjusted accordingly.

4.3.4 Computing Normal Forms of Postponed Equations

In every iteration of the inner loop of the version of Knuth-Bendix in Figure 4-2, an equation is selected and the normal forms of its two constituent terms are computed. Computing a normal form can be time-consuming. In the worst case, the left-hand side of every rewrite rule must be matched against each subterm of the term being reduced. In Section 6.2.2, we discuss the efficient computation of normal forms in general. Here, we present REVE's strategy for normalizing postponed equations.

Before REVE postpones an equation, it replaces the equation by its normal form. When the equation is reconsidered later, it is already in normal form with respect to whatever rewrite rules were in the rewriting system when it was previously normalized. The equation can only be reduced further if it is reducible by a rewrite rule that has been added to the rewriting system since the last time its normal form was computed. If the equation can be reduced using one of the new rules, the entire rewriting system must be used to find the new normal form; otherwise, the equation is already in normal form.

Though perhaps the most time-efficient strategy, it is probably prohibitively space-consuming to associate, with each normalized equation, the list of rewrite rules with respect to which the equation was normalized. Instead, REVE does the following: When an equation is ordered, the new rewrite rule is temporarily stored on a list of *unused* rules in addition to being added to the rewriting system. Before attempting to order any equations, REVE removes each rewrite rule from the unused list, and attempts to reduce each of the remaining postponed equations in the system with respect to that rule. Those equations that can be reduced by the unused rule are then changed into new equations, since they must be re-normalized using the entire rewriting system. No such normal form computation is necessary for the other equations. In this way, all equations are maintained in normal form with respect to the rewriting system minus the unused rules.

4.3.5 Knuth-Bendix Tasks and Organization

In the original Knuth-Bendix procedure (Figure 2-7, Page 29) there is a main loop that consists of finding a non-joinable critical pair, computing its normal form, ordering it, and normalizing the rewriting system. In Huet's version (Figure 4-2), there is an inner loop that processes all of the equations, and an outer loop that computes more critical pairs once all the equations have been processed. Knuth-Bendix would remain correct if we instead computed critical pairs in the inner loop, and only converted equations to rules in the outer loop, when the critical pairs had been exhausted. However, it is implicit in the procedure's formulation that the computing of critical pairs is a less "desirable" task than ordering equations. Indeed, critical pairs are expensive to compute, and one hopes that by first processing as many equations into rewrite rules as possible, many rules and equations will disappear when rewritten with the new rules, and thus fewer critical pairs will need to be computed.

REVE's Knuth-Bendix is more robust than Huet's, but also more complex. Consequently, there are several tasks, with some being more desirable than others. REVE's Knuth-Bendix tasks are shown in Figure 4-4, in decreasing order of desirability.

Determining the relative desirability of Knuth-Bendix tasks involved a subjective judgement. The overall goal was to maximize the set of systems that could be completed while minimizing user interaction and computation time. The assumptions underlying our desirability criteria are given in Figure 4-5. The Knuth-Bendix implementation in REVE was designed so that tasks can be re-ordered easily, if the underlying assumptions change. For example, Section 4.4 suggests a different order of tasks that is more appropriate when using automatic orderings.

Figure 4-6 shows the flow of control in REVE's Knuth-Bendix implementation. The tasks are performed in decreasing order of desirability, directly reflecting the assumptions of Figure 4-5. One hopes that more desirable tasks will reduce the work required of less desirable tasks; e.g., reducing equations to normal form may cause an unorientable equation to disappear.

The Knuth-Bendix interrupt and undo facilities in REVE are organized around the tasks. When the user interrupts Knuth-Bendix, REVE remembers the name of the current task so that Knuth-Bendix can later resume from that point. Undo is implemented by saving "pointers"[15] to all current rewrite rules and equations, plus the name of the current task, on a "history stack" just before each user interaction. When the user performs an undo, the history stack is popped and the current equations, rewrite rules, and task name get replaced by those that are referenced on top of the stack. (See Section 5.4.3).

[15]We have placed "pointers" in quotes, because CLU is object-oriented, and every object is always referenced via a pointer.

Figure 4-4: Tasks Performed by REVE's Knuth-Bendix Implementation

ReduceEquations: Remove a rewrite rule from the unused list, and attempt to reduce every postponed equation using that rewrite rule. Move, to the list of new equations, all equations that get reduced. Repeat with each unused rule until none remain.

ConsiderNew: Remove an equation from the new list, and reduce it to normal form with respect to the rewriting system. If the resulting equation is big, move it to the list of big equations. Otherwise, execute the algorithm in Figure 2-9 on Page 33. If the algorithm divides the equation into a set of new equations, add those equations to the new list. Otherwise, attempt to order the equation. Put the equation into one of 1) the list of unused rules and the rewriting system, 2) the list of incompatible equations, or 3) the list of unoriented equations, as appropriate. If the equation becomes a rule, normalize the rewriting system as per the procedure in Figure 4-2. Any rules that become equations as a result of normalization get added to the list of new equations. Repeat until there are no more new equations.

ConsiderIncompatible: Remove an equation from the incompatible list, and ask the user whether he wishes to divide or postpone the equation. Repeat until an equation has been divided or all incompatible equations have again been postponed.

CriticalPairs: Mark the smallest unmarked rule in the rewriting system, compute critical pairs between it and all marked rules, including itself, and add the critical pairs to the list of new equations. If no critical pairs result, repeat. If there are no unmarked rules, do nothing.

ConsiderUnoriented: Remove an equation from the unoriented list, and present the user with any suggestions, provided by the ordering's implementation, for extending the ordering. Ask the user to choose one of the actions shown in Figure 4-3. If the equation gets divided, add the two new equations to the new list. Repeat until a new equation or rewrite rule has been generated, or all unoriented equations have again been postponed.

ConsiderBig: Remove the smallest equation from the list of big equations, and adjust β so that the equation is no longer big. Process the equation in the same manner as for new equations in the *ConsiderNew* task (except that it is already in normal form). Repeat until a new equation or rewrite rule has been generated, or there are no more big equations.

ConsiderDeferred: Remove an equation from the deferred list. If the equation is incompatible, process the equation in the same manner as in the *ConsiderIncompatible* task. Otherwise, process it as in the *ConsiderUnoriented* task. Repeat until a new equation or rewrite rule has been generated, or all deferred equations have again been postponed.

Figure 4-5: Assumptions that Determine the Relative Desirability of Tasks

(1) Reducing equations to normal form is relatively cheap. It is also useful: as a result of this task, equations and rewrite rules can become smaller or disappear entirely.

(2) Ordering equations (without user help) can result in more rewrite rules, which in turn can allow other equations or rules to be reduced. The benefits are not as direct as for reducing equations, but it is not computation-intensive.

(3) An incompatible equation cannot be ordered, and user help is required to decide whether the equation should be divided into two. Nevertheless, dividing an incompatible equation can be very beneficial. Each of the rewrite rules that come from the two resulting equations has at least one variable on its left-hand side that does not occur on its right. Consequently, when either of these rules is used to reduce a term, one or more subterms of that term are effectively eliminated during the reduction. Incompatible equations occur infrequently, but their presence usually indicates an important underlying property of the equational theory that should be immediately incorporated into the completion process.

(4) Computing critical pairs can be time-consuming. Critical pairs must be ordered before they can be of further use, so they only contribute indirectly to the reducing of other equations. However, critical pairs are computed without user help.

(5) To order an unoriented equation, user help must be solicited to extend the ordering (if possible) or postpone the equation. Because of the user interaction, this task is not as desirable as the above tasks in the context of automatic theorem proving.

(6) As mentioned previously, big equations are rarely helpful to Knuth-Bendix. It is more desirable to consider the unoriented equations first, even though user help is required, because they are smaller.

(7) It is unlikely that any deferred equation is orderable. If there are any other equations, all of them should be ordered or divided before the deferred equations are considered, with the hope that the deferred equations will reduce. Thus, consideration of the deferred equations is the least desirable task.

Figure 4-6: Flow of Control in REVE's Knuth-Bendix Implementation

```
while there are any equations do
    ReduceEquations
    ConsiderNew
    if there are any unused rewrite rules then repeat endif
    ConsiderIncompatible
    if there are any new equations then repeat endif
    CriticalPairs
    if there are any new equations then repeat endif
    ConsiderUnoriented
    if there are any new equations then repeat endif
    ConsiderBig
    if there are any unused rewrite rules or non-deferred equations then repeat endif
    ConsiderDeferred
endwhile
```

4.4 Knuth-Bendix Using Automatic Orderings

As noted in Section 4.1, when automatic orderings are used with Knuth-Bendix, it is usually most efficient to try to order new equations before computing critical pairs, because earlier equations are generally smaller and more useful than later ones, and automatic orderings present less overhead in finding an appropriate registry extension to order an unorderable equation. This optimization for automatic orderings can be achieved in failure-resistant Knuth-Bendix by transposing the order of the *CriticalPairs* and *ConsiderUnoriented* tasks in Figure 4-6.

In Section 3.6, we presented a procedure, AutomaticConstruction, for automatically constructing a terminating rewriting system from a set of equations, which makes use of an automatic ordering. This could be used in Knuth-Bendix, when an automatic ordering has been chosen, by replacing most of the user interactions of Section 4.3.2 with the actions indicated by AutomaticConstruction. The history stack must also be augmented to store the additional information required by AutomaticConstruction.

Note that this does not make Knuth-Bendix, itself, fully automatic. AutomaticConstruction might initially choose a particular orientation for an equation that causes the completion process to diverge, with Knuth-Bendix generating an infinite set of critical pairs that are all orderable. AutomaticConstruction will not back up to reverse that equation in this case; the procedure is designed to work with a finite set of equations. See Section 6.2.4.2 for a discussion of implementing a fully-automatic Knuth-Bendix.

REVE currently provides an implementation of AutomaticConstruction that converts a fixed set of equations into a terminating rewriting system (when possible), but, in the context of the completion process, the registry is not extended automatically. Instead, when using an automatic ordering with Knuth-Bendix and an unorderable equation is encountered, the minimal complete extender set is presented to the user, and the user selects one of the minimal extenders (if any) to make the equation orderable.

Chapter Five

The REVE Program

5.1 Introduction

This chapter describes the REVE term rewriting system generator, a program for theorem proving and experimentation that is based on the material in the previous chapters. REVE was designed and implemented by the author and others. It is operational and continues to be improved; see the Preface on Page 4 for an overview of the development effort.

REVE supports equational and inductive theorem proving, and direct access to basic rewriting, unification, and superposition operations. Theorem proving is accomplished with the failure-resistant Knuth-Bendix implementation described in Chapter 4, which incorporates some support for Huet-Hullot inductionless induction. REVE currently includes an automatic ordering implementation of EPOS, and an implementation of [illegible] that provides suggestions (see Section 3.3.2), for use by Knuth-Bendix. In addition, individual terms can be rewritten and unified, and critical pairs can be computed for individual pairs of rewrite rules, for experimentation purposes.

The remainder of this chapter consists of three parts. Section 5.2 summarizes the features provided by REVE's user interface. Section 5.3 outlines some sample experiments one might perform with REVE. Finally, Section 5.4 presents the salient features of REVE's internal structure.

5.2 User Interface of REVE

REVE provides a robust, user-friendly, line-oriented user interface, intended to be suitable for both experienced and novice users. REVE's parser accepts the conventional notation for equations, rewrite rules, and terms, including infix operators. Commands may be typed in upper and/or lower case, and unambiguous prefixes are accepted. The user may type a command and some or all of its arguments all on one line. The user is explicitly prompted for any omitted arguments. Whenever REVE expects input, the user may type "?" to see the list of possible responses in the current context. The HELP command provides on-line documentation for each command, plus additional information on more general topics related to REVE's use.

REVE's remaining commands fall into the following categories:

- Handling the input, output, display, and deletion of the rules and equations manipulated by Knuth-Bendix.
- Selecting the registered ordering to be used by Knuth-Bendix, and controlling the precedence and status map to be used by that ordering.
- Invoking Knuth-Bendix and theorem proving.
- Directly accessing rewriting and unification primitives.
- Saving and restricting terminal input/output.

The remainder of this section presents an overview of these capabilities. See the Appendix on Page 116 for a detailed description of each command currently available in REVE.

5.2.1 System

The user's current system of rules and equations may be read from, and written to, disk files and the user's terminal. Individual rules and equations may be deleted from the system, and the user will be warned if such deletion might compromise the correctness of Knuth-Bendix.

In addition, the current system may be stored and retrieved in raw CLU object form[16]. When a system has been fully or partially completed by Knuth-Bendix, the user may FREEZE, into a file, the entire system state, including all of the current rules and equations, the current registry, and the "undo" history stack. Later, the user may THAW the frozen system. FREEZE and THAW are particularly useful for saving completed systems that are of general utility, or for temporarily saving an incomplete session with Knuth-Bendix.

[16]This idea has been borrowed from Affirm [Musser 80a].

5.2.2 Ordering and Registry

An ORDERING command is provided that allows the user to choose between EPOS and EDOS for the ordering that will be used by Knuth-Bendix. As indicated previously, EPOS computes the complete set of minimal extenders when an equation cannot be ordered, and EDOS currently computes $\gg$ suggestions. Alternatively, the user can select the "manual" ordering, which causes REVE to present each equation to the user so that it can be hand-ordered.

Normally, the user extends the current registry incrementally as each unorderable equation is considered by Knuth-Bendix. However, REVE commands for initializing, extending, and viewing the current registry are also provided by the top-level command interpreter.

5.2.3 Knuth-Bendix and Proofs

The KB command invokes Knuth-Bendix on the current system. Knuth-Bendix can be interrupted at any time by typing ↑G (control G). The user can then invoke other commands, and subsequently continue the completion process. At any time, UNDO (Section 4.3.2) can be invoked one or more times to return to any previous user interaction (e.g., to choose a different minimal extender for an equation or to divide an incompatible equation), and Knuth-Bendix can be resumed from that point.

Equational and inductive proofs are performed with PROVE. PROVE takes an equation as its argument, and attempts to prove that the equation is in the equational or inductive theory of the current system. PROVE first uses the current rewriting system to reduce the equation to normal form; if the two sides of the equation become equal, the theorem holds. Otherwise, if the current system has not been completed by Knuth-Bendix, Knuth-Bendix is automatically invoked (after user confirmation). If Knuth-Bendix terminates successfully, the equation is again normalized. If the two sides are equal, the equation is valid in the equational theory. Otherwise, after user approval, REVE automatically checks to see if the equation is in the inductive theory: the equation is added to the current system, and Knuth-Bendix is again invoked. If the procedure completes successfully, the user is told that the equation is an inductive theorem. If the procedure aborts with Huet-Hullot *pseudo-inconsistency*, the equation is invalid in the inductive theory.

For Huet-Hullot inductionless induction to be sound, the user must declare HH-constructors using the HH-CONSTRUCTORS command prior to running Knuth-Bendix, and the system must be shown to satisfy the principle of definition with respect to these constructors. As noted in Section 2.7, this condition is undecidable, and REVE does not yet check for sufficient conditions. Currently, it is the user's responsibility to ensure that the definition principle holds.

5.2.4 Basic Operations

Basic rewriting primitives are invoked with the REDUCE and NORMAL-FORM commands. Both of these commands operate on a term given by the user. REDUCE reduces the term (if possible) once, using an arbitrary applicable rewrite rule from the current rewriting system. NORMAL-FORM computes the normal form of the term with respect to the current rewriting system, and also displays all intermediate reduced forms. If the term gets rewritten an inordinately large number of times and no normal form has yet been found, REVE assumes that rewriting will probably not terminate. In this case, the normal form computation stops, and the user is shown the last several intermediate reduced forms to help in identifying the source of the non-termination.

The UNIFY and CRITICAL-PAIRS commands permit access to the primitive operations used by Knuth-Bendix. The UNIFY command accepts two terms as arguments, and displays their unification, or indicates that the two terms cannot be unified. The CRITICAL-PAIRS command displays all the critical pairs, if any, that result from superposing two rewrite rules given by the user.

5.2.5 Terminal Session

The last category of commands control the terminal session itself, and are provided for user convenience. These commands are fairly independent of the application domain; they do not directly pertain to the rewriting and theorem proving capabilities of REVE.

Two commands allow terminal interaction to be sent to a file at the same time it is seen on the screen. The SCRIPT command takes a file name, and sends all terminal input/output to that script file for later viewing. The LOG command causes all terminal input (only) to be stored in a log file. The log file can later be "played back" using the REPLAY command. REPLAY causes REVE to take all user input from a log file, and returns control to the user's terminal when the log file is exhausted. This feature is useful for doing demonstrations and for regression testing. UNSCRIPT and UNLOG close the current script file and log file. Also, these files are automatically closed when the user exits from REVE.

The TRACE command controls the amount of output produced by Knuth-Bendix. REVE's Knuth-Bendix implementation is capable of displaying many kinds of information during the completion process, including:

- The next equation under consideration.
- The normal form of that equation.
- The rewrite rule that comes from that equation.
- The critical pairs that result from the new rule.
- The rewrite rules whose right-hand sides are rewritten by the new rule.
- The rewrite rules whose left-hand sides are rewritten by the new rule.
- The equations that result from rewriting left-hand sides of rules.

Inundating the user with all of the above information, although useful in some experimentation contexts, is not always desirable. Accordingly, a TRACE command is provided, which provides various levels of output from Knuth-Bendix. The highest tracing level displays all of the above information. The lowest level displays nothing, unless user input is required. (This level is particularly useful when REVE is serving as an embedded theorem prover.) Intermediate tracing levels fall between these extremes. Though it currently only pertains to Knuth-Bendix, TRACE is intended to be a generic output control facility that will be enhanced to work with future REVE facilities as they are incorporated.

Many REVE activities can produce more information than can fit on a single terminal screen (Knuth-Bendix, displaying the current system, showing all intermediate terms leading to a normal form, etc.). For this reason, an optional "page mode" is provided that stops REVE output after a screen of text has been displayed, and waits for a response from the user. The user may direct REVE to display the next full or half screen of output, display some additional number of lines, continue displaying until the next user interaction, or stop displaying until the next user interaction. Alternatively, most operating systems provide some separate means for controlling output[17]. However, these capabilities are sometimes inoperable when the computer is accessed over a network from a remote host, so REVE's page mode feature can be particularly useful for remote users.

5.2.6 Possible Enhancements

REVE's line-oriented user interface provides on-line help facilities, a robust parser, and a flexible command interpreter. Many enhancements are possible, however, to extend its functionality and ease of use. This section presents some of the user interface improvements that are under consideration.

The user interface could benefit from many features found in screen-oriented text editors. Multiple windows could be established, to allow the user to cut and paste, view, and scroll both input and output. Separate windows could also be established for the current rewriting system and set of equations, enabling the user to dynamically view the system changes effected by Knuth-Bendix.

There are many useful statistics that might be collected about a Knuth-Bendix run and provided to the user, to measure the complexity of examples, to identify REVE modules where efficiency optimizations are needed, etc. The original Knuth-Bendix paper [Knuth 70] used an "efficiency rating" — a ratio of "useful" derived rules to the total number of derived rules — to measure the effectiveness of the procedure. Other statistics possibilities are the number of rewrites, number of unifications, average ratio of number of equations to number of rules, largest number of equations at any one time, number of critical pairs, average number of rewrites required when normalizing a term, size of largest critical pair, number of user interactions required, time spent in rewriting, time spent in unification, time spent in ordering equations, and total time spent in completing the system.

Type checking has been found to be a useful facility when developing large programs. Similarly, sort[18] checking can be useful when using large sets of equations in REVE. REVE

[17] For example, when REVE runs under Unix, ↑S and ↑Q can usually be typed by the user to stop and start output.

[18] *Sorts* in algebra are analogous to *types* in programming languages.

already performs some limited checking, to ensure that the arity of an operator is the same throughout the system. A complete sort checking scheme would ask the user for the signature (domain and range sorts) of each operator in the system, check that the range sorts of arguments match the corresponding domain sorts of root operators in all input terms, and ensure that the two terms comprising each input rewrite rule and equation have matching sorts. Mandayam Srivas and his students at the University of New York at Stony Brook have designed and implemented such a sort checking scheme in an experimental version of REVE.

Once sort checking of new rules and equations is complete, the sort information is no longer needed, since the procedures and algorithms in REVE preserve sort correctness. Thus, it might appear that sort information should be confined to the user interface, or to application programs that use REVE. On the other hand, it is possible that this information could be used to improve the efficiency of various algorithms in REVE.

5.3 Examples of REVE's Use

Chapters 2, 3, and 4 presented the underlying rewriting theory and procedures embodied in REVE. The above section gave an overview of REVE's commands and capabilities. In this section, we bring together theory and practice by presenting two concrete examples using REVE. These are toy examples, but they illustrate the program's salient features and the flavor of typical terminal sessions.

First, we will show how REVE might be applied to the popular group theory example. This will include running the Knuth-Bendix procedure, and proving an equational theorem. The particular commands used to accomplish each task will be indicated.

Second, we will describe how REVE could be used to reason about the theory of some equations that characterize the Fibonacci function. We will indicate a use for computing normal forms outside of Knuth-Bendix itself. An application of Knuth-Bendix inductionless induction will be presented. The example will show one reason to interrupt Knuth-Bendix and use REVE's "undo" facility. We will also describe how auxiliary lemmas can be introduced to assist REVE in proving an inductive theorem.

5.3.1 Group Theory Example

To begin the group theory example, we start up REVE, and use the READ command to input the axioms shown in Figure 5-1 (which are the same axioms as in Figure 2-1 on Page 19) from a previously-prepared file. Alternatively, we could invoke the TERMINAL command and type the group axioms directly. Either way, REVE responds by displaying the current contents of the system at the terminal. We use the ORDERING command to set the current registered ordering to be the automatic ordering EPOS.

Figure 5-1: Axioms for Group Theory

(1) $e \bullet x = x$

(2) $x^{-1} \bullet x = e$

(3) $(x \bullet y) \bullet z = x \bullet (y \bullet z)$

We then invoke the KB command to start the Knuth-Bendix procedure on the group axioms. REVE will display, among other things, each equation as it becomes ordered, and the critical pairs that get computed. Even though the current registry is empty, EPOS (because it is a simplification ordering) is able to order the first group axiom into the rule $e \bullet x \rightarrow x$. The two sides of the axiom $x^{-1} \bullet x = e$ are not orderable under the empty registry, however, so REVE presents the two minimal extenders "$^{-1} \gtrsim e$" and "$\bullet \gtrsim e$" to us, and we are told that either one will order the equation into a rewrite rule from left to right. We arbitrarily choose the first extender, REVE orders the equation, and Knuth-Bendix continues. We are prompted to select minimal extenders for two more unorderable equations during the completion process. All critical pairs are equational consequences of the original axioms. Along the way, we see various critical pairs that reveal that the left identity, e, is also a right identity; e is its own inverse; the left inverse, $^{-1}$, is also a right inverse; and $(x^{-1})^{-1} = x$ (which is Equation 1 on Page 18) is in the equational theory. When Knuth-Bendix completes, REVE prints the completed system shown in Figure 2-8 on Page 30.

The completed system gives us a decision procedure for group theory. We can now prove, for example, that $(x^{-1} \bullet y^{-1})^{-1} = y \bullet (x^{-1} \bullet e)^{-1}$ (which is Equation 4 on Page 30) is a theorem by using the PROVE command. This causes REVE to reduce both sides of the equation to normal form and compare the normal forms for equality. REVE indicates that the equation is, indeed, an equational theorem.

5.3.2 Fibonacci Function Example

In this section, we use inductionless induction to prove that two characterizations of the Fibonacci function, *fib* and *dfib*, are equivalent. [Lescanne 83a] contains a terminal session with REVE 1 on this example. The interested reader may wish to consult [Lescanne 83a] to compare the use of REVE 1 (there) with REVE 2 (here).

We read the equations shown in Figure 5-2 into REVE. The first two equations define addition in terms of the zero and successor functions of Peano arithmetic. The third equation is an inductive theorem of addition. We have introduced it as an axiom here, because we are interested in proving properties about *fib*, rather than +. The last three equations comprise the classical definition of the Fibonacci function, *fib*.

Figure 5-2: Equations Describing the *fib* Function

(1) $0 + x = x$

(2) $s(x) + y = s(x + y)$

(3) $(x + y) + z = x + (y + z)$

(4) $fib(0) = 0$

(5) $fib(s(0)) = s(0)$

(6) $fib(s(s(x))) = fib(x) + fib(s(x))$

We will want to use Huet-Hullot inductionless induction (see Section 2.7), so we use the HH-CONSTRUCTORS command to declare 0 and *s*, the constructors of nonnegative integers. It is our responsibility to declare appropriate HH-constructors, and to verify that our axioms satisfy the principle of definition with respect to those HH-constructors. We then invoke

Knuth-Bendix. We are asked to choose minimal extenders for two unorderable equations during the completion process. Knuth-Bendix finds no non-joinable critical pairs along the way, so the resulting convergent rewriting system contains just the original axioms (ordered).

Since the irreducible ground terms in this system are exactly those that consist solely of 0 and *s*, the normal form of *fib*(*n*) (where *n* is built with 0 and *s*) is the *n*'th Fibonacci number. Thus, we might use the NORMAL-FORM command at this point to find that *fib*(*s*(*s*(*s*(*s*(0))))) is *s*(*s*(*s*(0))).

We now add the three equations in Figure 5-3, which describe *dfib*, to the system. The equation

$$fib(x) = dfib(x, 0) \qquad (6)$$

directly expresses the meaning of *fib* in terms of *dfib*. We invoke PROVE to verify that this equation is a theorem of the above equations and rewrite rules. PROVE finds that both sides of Equation 6 are irreducible with respect to the current rewriting system, and thus the normal forms are not identical. The equation might still be an equational theorem, however, since the current system (consisting of the previously-completed convergent rewriting system and the equations in Figure 5-3) is not complete. Consequently, PROVE automatically invokes Knuth-Bendix, after user confirmation.

Figure 5-3: Equations Describing the *dfib* Function

(1) $dfib(0, y) = y$

(2) $dfib(s(0), y) = s(y)$

(3) $dfib(s(s(x)), y) = dfib(s(x), dfib(x, y))$

When considering the third equation in Figure 5-3,

$$dfib(s(s(x)), y) = dfib(s(x), dfib(x, y)) \qquad (7)$$

REVE presents the user with three minimal extenders: either $\psi(dfib)$ = Ⓜ or $\psi(dfib)$ = Ⓡ will order the equation from right to left, and $\psi(dfib)$ = Ⓛ will order the equation from left to right. We choose $\psi(dfib)$ = Ⓜ. Accordingly, REVE reverses Equation 7, converts it to a rewrite rule, and Knuth-Bendix continues. At this point, the completion procedure diverges: it starts generating an infinite set of ever-larger rules. In some cases, this difficulty can be averted by choosing a different orientation for a previous equation. We interrupt Knuth-Bendix by typing ↑G, and invoke the UNDO command, which backs up the completion process to the last user interaction. (We can perform successive UNDOs to return to any previous interaction.) In this case, we are again presented with Equation 7. This time, we choose the minimal extender $\psi(dfib) = ①$, and Knuth-Bendix completes successfully. It can be difficult, even for experienced users, to choose an appropriate minimal extender for an unorderable equation. This is one reason why the UNDO command is so useful.

This Knuth-Bendix run was actually performed as part of the PROVE command. Since Knuth-Bendix has successfully terminated, PROVE again checks whether Equation 6 is an equational theorem. It is not, so PROVE automatically uses inductionless induction (after user confirmation). This entails adding Equation 6 to the system, and running Knuth-Bendix once again.

After asking the user to pick minimal extenders for two unorderable equations, Knuth-Bendix diverges. Further experimentation would reveal that choosing other minimal extenders for the equations will not solve the problem. This situation can often be alleviated by first finding and proving an inductive lemma that may help in proving the theorem of interest. We interrupt Knuth-Bendix, cancel the proof with the CANCEL command, and attempt to prove the lemma

$$fib(x) + y = dfib(x, y) \qquad (8)$$

We hope that this equation, which is a more general version of Equation 6, may be easier for Knuth-Bendix to handle.

PROVE finds that Equation 8 is not an equational theorem of the system. Therefore, PROVE adds the equation to the system, runs Knuth-Bendix, and the system completes successfully. PROVE announces that Equation 8 is an inductive theorem of the system (though, as noted above, the soundness of this inductionless induction scheme requires that the initial system satisfy the principle of definition, which must be verified by the user). If the algorithm in Figure 2-9 on Page 33 had halted with *pseudo-inconsistency*, REVE would have told us that Equation 8 is not valid in the inductive theory of the system.

Having proven the lemma, we return to proving the original theorem of interest, Equation 6. This time, Knuth-Bendix completes, and the inductionless induction proof of Equation 6 is successful.

5.4 Internal Structure of REVE

This section gives an overview of the major modules in REVE, and how they interact. This information is primarily intended for programmers who wish to extend REVE or adapt it to their purposes. Throughout this section, names in **boldface** are module names in REVE's implementation. There are many minor and general purpose modules that are not discussed here; e.g., **set**, **list**, **mapping**, and **scanner**. Also, we omit discussion of modules that are used only by the orderings, the unification algorithm, and the user interface.

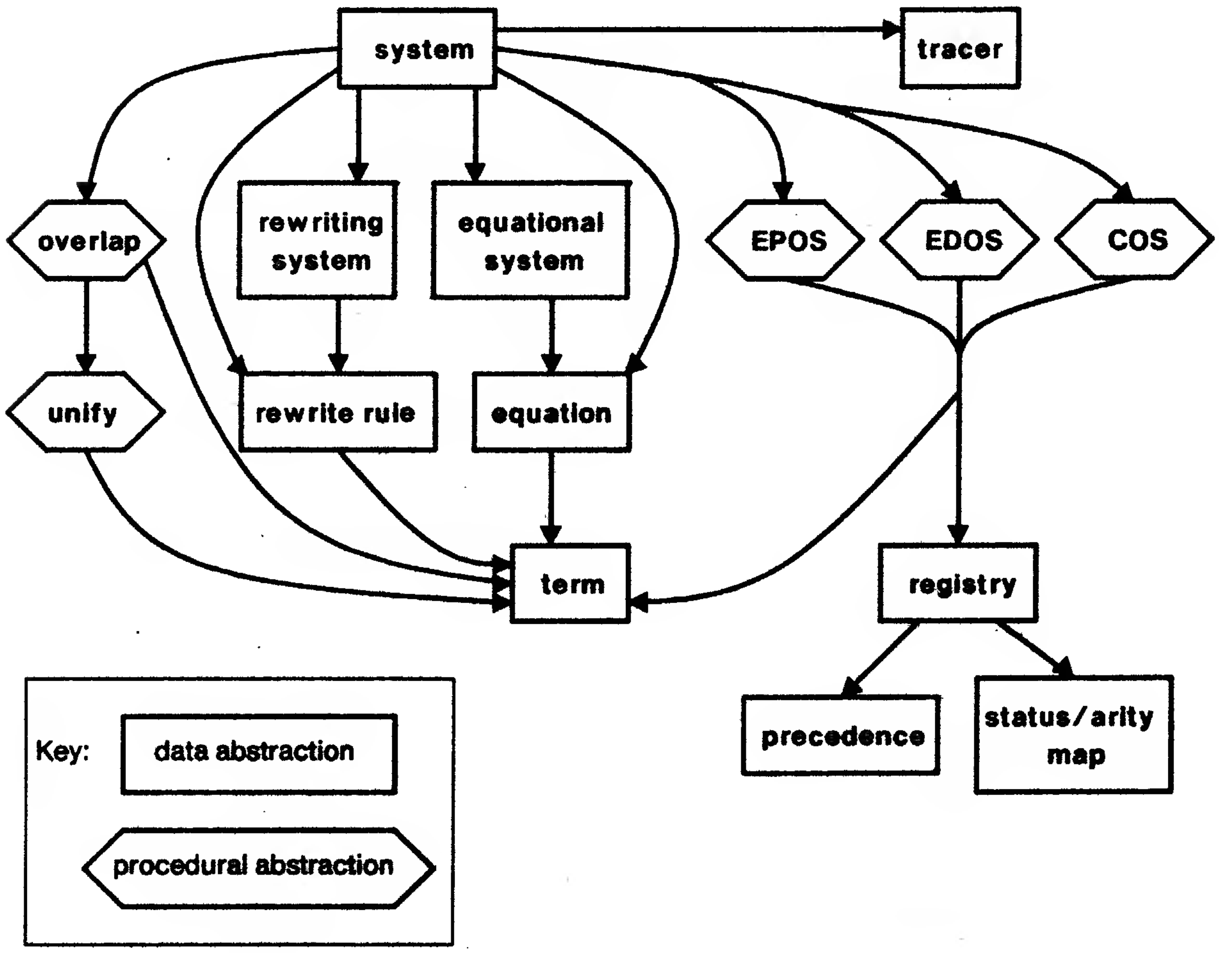
REVE is written in the programming language CLU [Liskov 81], which provides mechanisms for data abstraction (*clusters*), procedural abstraction (*procedures*), and control abstraction (*iterators*). In CLU, a *module* is either a cluster, procedure, or iterator. A cluster has a concrete representation type, called the *rep*, for the abstract type it implements, as well as a set of operations for manipulating objects of the abstract type. These cluster operations, which are themselves procedures or iterators, are the only means of manipulating objects of the corresponding abstract type. The abstract type implemented by a cluster may be 1) *mutable*, which means that the state (value) of objects of that type can be changed, or 2) *immutable*, which means that any object of that type, once created, always has the same state.

Figure 5-4 is a module dependency diagram for most of the clusters and procedures that we will discuss here. There is an arc from a module, A, to another module, B, if A directly uses B in its implementation.

5.4.1 Registry, Precedence, and Status/Arity Map

The **registry** stores all operator information needed by REVE. Like the mathematical notion of *registry* introduced in Section 3.2.3, a **registry** consists of a precedence and status map. In addition, the **registry** stores the arity of each operator. REVE uses the arity information to ensure that each operator always has the same arity in all terms that are read as input. In the future, if sort information is incorporated into REVE (see Section 5.2.6), the signature (domain and range sorts) of each operator will also be stored in the **registry**. The **registry** preserves the invariant that it be consistent (Section 3.2.3) with respect to its **precedence** and **status/arity map** components.

Figure 5-4: Module Dependency Diagram for the Major Modules in REVE



The **precedence** cluster is implemented as a labelled directed acyclic graph. The nodes are operators, and there are two kinds of edges: $\gtrdot$ and $\doteq$. If $f \;\underline{\gtrdot}\; g$, there is both a $\gtrdot$ and a $\doteq$ edge from f to g. The **precedence** maintains the invariant that it be consistent (Section 3.2.3).

The **status/arity map** is a mapping from operators to their status and arity. If an operator has not been assigned a status, its status is recorded as ⓤ.

5.4.2 EPOS and EDOS

The implementation of Knuth-Bendix is independent of the particular ordering being used. It requires only that an ordering module provide two procedures that attempt to order an equation: a *quiet* procedure that does not interact with the user, and a *user* procedure that may obtain user assistance. The quiet procedure is used by the *ConsiderNew* and *ConsiderBig* tasks (see Figure 4-4 on Page 81), and the user procedure is used by the *ConsiderUnoriented*, *ConsiderIncompatible*, and *ConsiderDeferred* tasks. Both procedures have access to the **registry**. The quiet procedure just returns the result of comparing the two terms, and does not change the **registry**. The user procedure may return the comparison, or may inform Knuth-Bendix that the user wishes to postpone the equation, divide it into two, interrupt Knuth-Bendix, or "undo." In addition, the user procedure may extend the precedence and status map in the **registry**.

EPOS and EDOS are the two orderings currently available in REVE. The quiet procedure in both of these modules merely checks whether an equation is currently orderable, in either direction, under that ordering. The user procedures of EPOS and EDOS compute minimal extenders and $\gg$ suggestions, respectively, for each unorderable equation. The COS module is shown in Figure 5-4 for illustrative purposes, to indicate how future ordering implementations will fit into REVE's internal structure.

5.4.3 Unify and Overlap

The procedure **unify** takes two terms and returns the most general unifier of those terms. The unification algorithm currently used in REVE is that of Martelli & Montanari [Martelli 82], whose efficiency compares favorably with other algorithms on typical examples.

The procedure **overlap** takes two rewrite rules, computes the superpositions associated with each overlap between the left-hand sides of the two rules, and returns all of the critical pairs resulting from those superpositions. This procedure is the heart of the confluence test in Knuth-Bendix.

5.4.4 Term, Rewrite Rule, and Equation

The term cluster implements the mathematical notion of term as an immutable object. Operations are provided for finding the size of a term, determining the set of variables and set of operators in a term, obtaining the subterm at a given occurrence in a term, and creating a new term that has the same structure as an existing term but with a different subterm at a given occurrence.

Terms are the most basic and often-manipulated objects in REVE, so it is important that their implementation be efficient. Since terms are immutable, the size, set of variables, and set of operators in a term cannot change over time. Thus, we need only compute these attributes once for any given term. Since one rarely needs all of these attributes for any particular term, REVE computes an attribute for a term the first time it is needed. It then keeps that attribute value in the rep of the term so that it need not be recomputed in the future. Note that this implies a mutable rep for terms, but this mutability is not observable from outside the term cluster, so the abstraction is immutable.

The rewrite rule and equation clusters correspond to the mathematical notions of the same names. Both types are immutable. There is a rewrite rule operation for reducing a term once with respect to a rewrite rule. The immutability of rewrite rules and equations is useful for the history stack mechanism. When placing a "copy" of the current system on the stack, only "pointers" to the current rewrite rules and equations are needed, because there is no danger that the state of these particular rules and equations will change during subsequent processing.

5.4.5 Rewriting System, Equational System, and System

Rewriting system implements the mathematical notion of a term rewriting system. A rewriting system is mutable so that rewrite rules can be added to it. Operations are available for reducing a term once with respect to the rewriting system, and for computing the normal form of a term. A means is provided for terminating an excessive number of rewrites during a normal form computation if the rewriting system is not known to terminate.

Internally, a rewriting system is divided into marked and unmarked rewrite rules, to support the marking scheme for choosing rewrite rules during critical pair computations (see Section

4.3.1). There is a special **rewriting system** operation that returns a list of all marked rewrite rules plus the smallest unmarked rewrite rule, and marks the latter. Knuth-Bendix computes critical pairs between the latter rule and all the marked rules.

To improve performance, REVE borrows an idea from Affirm [Musser 80a]: Stored in the rep of the **rewriting system** is a hash table that maps operators to buckets of "pointers," where each "pointer" points to a rewrite rule in the marked list or the unmarked list. The root operator of the left-hand side of each rewrite rule serves as the hash key for that rule. When reducing a term or subterm, $t = f(\ldots)$, the rewriting operation only needs to try the rules referenced by the bucket associated with f. Rules not referenced by "pointers" in that bucket will not match t.

An **equational system** consists of all equations to which Knuth-Bendix is being applied. The equations in an **equational system** are divided into five lists, as described in Section 4.3.3: new, unoriented, incompatible, deferred, and big. Special operations are provided for manipulating these lists and for computing the current value of β.

A **system** contains a **rewriting system** and an **equational system**. Its key operation is the failure-resistant Knuth-Bendix procedure described in Chapter 4. By encapsulating the Knuth-Bendix equations and rewrite rules within a single **system** data abstraction, and thus controlling access to the data being manipulated by Knuth-Bendix, the integrity of the completion process can be maintained. Also contained in the rep of a **system** are:

- The ordering being used by Knuth-Bendix.
- The list of unused rewrite rules.
- The set of HH-constructors.
- The name of the Knuth-Bendix task currently being executed. This is used when the user interrupts Knuth-Bendix, and later asks REVE to resume completing the system.
- The history stack, used to implement the "undo" facility in Knuth-Bendix.
- Total Knuth-Bendix running time for the current system, less all time lost along decision paths that were subsequently cancelled with "undo."
- The **tracer** (see the next section).

5.4.6 Laboratory and Tracer

The **laboratory** operations correspond to all of the useful functions available in REVE that are not related to the user interface. Together, they form a rewrite rule laboratory. (The **laboratory** cluster is not shown in Figure 5-4. It uses almost all of the modules in the figure.) A user interface to REVE need not make direct use of any modules below the **laboratory**. Applications that wish to use REVE's capabilities can be built directly on top of the **laboratory** cluster.

At the present time, there is only one user interface to REVE. This interface reads input from the user's terminal or from files, invokes the desired **laboratory** function, and prints the results on the user's screen. Most user interaction is orchestrated directly by this user interface module.

There are some interactions with the user for which it is not convenient to use the top-level user interface. Prominent among these are the informational messages printed by Knuth-Bendix, and the choosing of minimal extenders by the user. For these situations, the **tracer** module is provided. All modules below the level of **laboratory** perform all of their input and output to the terminal through **tracer**. **Tracer** provides a different procedure for each possible type of output message produced by REVE. Although the **tracer** supports various levels of output (see Section 5.2.5), this feature is invisible to the modules that use it: **tracer** merely filters out those display messages that are not appropriate for the current tracing level. If desired, the **tracer** module implementation can be easily changed to support a different style of user interface.

Chapter Six

Summary and Conclusions

This chapter presents a summary of the thesis, indicates some areas of future implementation and research, and reflects on the development of REVE.

6.1 Summary of Contributions

In this thesis, and the associated implementation work, the author has:

- Presented the basic theory of term rewriting, and equational and inductive proofs, in a manner that should be accessible to computer scientists who are not familiar with the area.

- Developed a method for automatically constructing a terminating rewriting system from a set of equations. This method, based on simplification orderings, uses new algorithms that compute minimal complete extender sets for unorderable terms. The orderings supported by the method include improved, fully extensible versions of existing orderings, and a recent closure ordering.

- Designed and implemented a new failure-resistant version of the Knuth-Bendix completion procedure, particularly well-suited to automatic theorem proving applications. It features a strategy for automatic postponement of unorderable equations that considers "easier" equations first, an "undo" facility that can back up the completion process to change the response at any previous decision point, and support for the Huet-Hullot "inductionless induction" method.

- Designed and implemented most of REVE 2, a production-quality program that incorporates the above ideas in a powerful, user-friendly system that is suitable for theorem proving and experiments in term rewriting. The REVE source code is modularly designed and carefully documented, in the hope that it may provide the basis for experimental implementations in this area by other researchers, and thus expedite the development process.

6.2 Current Limitations and Ideas for the Future

REVE continues to be enhanced, both with new features and fine tuning. We list here some of the improvements that are either under development or under consideration.

6.2.1 A Rewrite Rule Laboratory

A primary goal of REVE 2 is to provide a solid source code base upon which one can easily build implementations of experimental programs in the rewriting area. Unfortunately, since REVE is written in CLU, making changes or additions to REVE requires some recompilation. As explained below in Section 6.3, we feel that CLU's advantages outweigh this disadvantage. Nevertheless, it is worthwhile looking at another software system, RRL, that provides many of REVE's features in an interpretive language environment.

Kapur & Sivakumar's Rewrite Rule Laboratory (RRL) [Kapur 84a] is an environment for experimenting with algorithms for manipulating term rewriting systems and equational theories. Its goals differ from those of REVE 2, primarily in that RRL emphasizes easy experimentation and de-emphasizes automatic theorem proving. Accordingly, RRL has been written partially in Musser's interpreted language, L [Musser 84], and partially in LISP. L is based on LOGO and LISP and has been designed with the RRL application in mind. L will also serve as the command language for RRL and the language in which a user can interactively program small experiments. To build on RRL or change an existing function, the user need only type in a new or replacement function, written in L. No recompilation is necessary.

RRL currently lacks REVE's scheme for constructing terminating rewriting systems automatically, and the failure-resistant Knuth-Bendix implementation. Conversely, REVE currently lacks many of RRL's facilities for experimentation, such as different rewriting/normalization strategies (see Section 6.2.2, below), different strategies for computing critical pairs, and different unification algorithms. Both the REVE project and the RRL project have profitted in the mutual exchange of information and ideas between our respective research groups.

6.2.2 Rewriting

Term rewriting is the heart of REVE. This section presents methods for improving rewriting efficiency and extending REVE's rewriting capabilities. See also Section 6.2.4.3, where equational rewriting is discussed.

In a simple approach to rewriting, reducing a term with respect to a rewriting system might require matching each subterm of that term with the left-hand side of each rewrite rule in the rewriting system. As noted in Section 5.4.5, REVE uses Affirm's [Musser 80a] hash table idea to increase rewriting speed.

Affirm also uses pattern-match compilation (PMC) [Guttag 78b] to improve the efficiency of performing reductions. In PMC, all rewrite rules with the same root operator on the left-hand side get compiled into a single LISP function that reduces any term that has that root operator. If the rewriting is successful, this function calls the appropriate function to further reduce the rewritten term. The LISP functions are stored in a hash table (a LISP a-list), where the root operator is the hash key, as described above. This idea cannot be directly implemented in REVE; CLU is a compiled language, so CLU functions cannot be both created and invoked while REVE is running. However, each LISP function could probably be closely simulated with a special data structure, call it a *multi-rule*, that represents all rules in the rewriting system that have a given root operator on the left-hand side. It is likely that a fast interpreter for multi-rules could be written in CLU.

Plaisted [Plaisted 83] has advanced an idea to speed up normal form computations. He suggests associating a hash table with the rewriting system, where the hash keys are terms and the values stored in the hash table are rewrite rules. Whenever the normal form, t_2, of a term, t_1, is found, one adds the rule $t_1 \rightarrow t_2$ to the hash table under the hash key t_1. When computing the normal form of a term, t_3, first hash t_3 and try to match t_3 against the left-hand sides of each rewrite rule in the resulting hash bucket. If a match is found, rewrite t_3 using that rewrite rule, and then compute the normal form of the resulting term with respect to the rewriting system. In this way, several reduction steps can often be skipped.

REVE uses a "leftmost-outermost" strategy to rewrite a term, t: it attempts to rewrite t at its root using each of the rules in the rewriting system. If this is unsuccessful, REVE then attempts to rewrite each of the immediate subterms of t using each rewrite rule, and so on.

Kapur & Sivakumar have compared this strategy with three others [Kapur 84a], in the context of computing normal forms. Each of these four strategies has been implemented in RRL. In their experiments, Kapur & Sivakumar have found one strategy that is often faster than leftmost-outermost. It is a modification of leftmost-innermost that recognizes that certain subterms have already been normalized. As in RRL, it may be useful to allow the user to choose from among these strategies for experimentation purposes. Also, if the modified leftmost-innermost strategy is found to be faster than leftmost-outermost on most typical examples, it may be worthwhile using the former strategy as the default in REVE.

Additional expressive power for equational specifications can be obtained by associating a Boolean condition with each equation. The semantics of a conditional equation are that the equation holds whenever the condition is true. For example, with an equation that defines division, one might associate a condition that the divisor be non-zero. Such a specification can be converted into a conditional rewriting system, where a rewrite rule can only be used for rewriting if its condition is true for the term being rewritten. The conditions associated with the rewrite rules also affect the proof of termination and the Knuth-Bendix completion procedure. Zhang has implemented, using REVE's modules, the prototype of a program for validating conditional specifications using conditional term rewriting techniques, based on his work with Remy [Remy 84, Remy 85]. This work will be incorporated into ECOLOGISTE [Barros 84], a structured specification support system.

6.2.3 Simplification Orderings

The simplification orderings used in REVE are instances registered orderings, all of which are descendants of Dershowitz' recursive path ordering. This section describes a method for extending registered orderings further, and presents a simplification ordering that is not parameterized on registries.

In some cases, before using a registered ordering to construct a terminating rewriting system from a set of equations, it may be useful for the user to designate a particular constant operator in the system as being the *least constant*. This constant is, by definition, less than or equal to every other operator in the precedence. Thus, under RPOS, EPOS, EDOS, and COS, it is also less than or equal to every term that consists only of operators in the system. The definitions of these registered orderings can be extended to use this information, by considering the least constant to be less than or equal to every variable. This allows these orderings to prove the termination of additional rewriting systems[19]. Isabelle Gnaedig of the Centre de Recherche en Informatique de Nancy (CRIN) has implemented the least constant extension of RPOS in an experimental version of REVE. Francoise Bellegarde, also of CRIN, has found this feature to be important in her use of REVE to prove theorems about FP [Backus 78] programs [Bellegarde 84], taking the identity function to be the least constant.

Registered orderings are among the most commonly-used classes of simplification orderings. Another relational in popular use, which is not parameterized on registries, is the *polynomial ordering*.

Lankford [Lankford 79a] and Dershowitz [Dershowitz 79b] have suggested associating a polynomial, $F(\alpha_1, ..., \alpha_n)$, with each n-ary operator, f, in the system. This mapping extends to a morphism, μ, on terms by letting $\mu(f(t_1, ..., t_n)) = F(\mu(t_1), ..., \mu(t_n))$. The *polynomial ordering*, $\succ[\mu]$, on the relation is defined as $s \succ[\mu]\, t$ if and only if $\mu(s) > \mu(t)$ for all assignments, $\mu(x)$, to the variables in s and t.

Note that $\succ[\mu]$ is a partial ordering. However, $\succ[\mu]$ is not necessarily a simplification ordering. For any given μ, compatibility and the subterm property must be shown separately. Dershowitz suggests using polynomials over the real numbers. In this context, for any rewriting system, $\mathfrak{R}$, it is decidable [Tarski 51] whether there exists a μ such that $\succ[\mu]$ is a simplification ordering that proves the termination of $\mathfrak{R}$. However, this is not yet a practical method for proving termination, since existing decision procedures [Cohen 69] require super-exponential time.

Lankford suggests restricting the polynomials to those over the positive integers. All such polynomials have the compatible and subterm properties, so $\succ[\mu]$ is a simplification ordering in this setting. However, for positive integer polynomials, it is undecidable whether there exists a μ for $\mathfrak{R}$ such that $\succ[\mu]$ proves the termination of $\mathfrak{R}$. Nevertheless, for a proposed μ, it is often possible to check, by hand, whether $\mu(s) > \mu(t)$ for all assignments to the variables, and for every rule $s \rightarrow t$ in $\mathfrak{R}$. This is typically accomplished, for each rule, by factoring the polynomials $\mu(s)$ and $\mu(t)$, and dividing out the common factors. (Such dividing is permitted because no factor is equal to zero, since the polynomials range over positive numbers.)

[19]For example, the termination of $\{f(g(x)) \rightarrow g(a)\}$, where $g \gg f$, can be proven using the least constant extension of any of these orderings, if a is the least constant.

When registered ordering implementations are not available, the polynomial ordering (using polynomials over the positive integers) is sometimes easier to use than registered orderings when proving termination by hand. However, the main reason for incorporating the polynomial ordering into REVE is that there are rewriting systems whose termination cannot be proven with existing registered orderings, but can be proven with the polynomial ordering[20]. The converse is also true [Dershowitz 83c], so both registered orderings and the polynomial ordering should be provided in REVE. Implementing the polynomial ordering will not be easy. It is difficult to develop procedures for comparing polynomials, and for automatically deriving an appropriate μ for a given $\mathfrak{R}$, that are sufficiently powerful to be generally useful. Lescanne and Alhem Bencheriffa are studying these problems for REVE.

6.2.4 Completion Procedure

REVE derives its theorem proving capabilities from the Knuth-Bendix completion procedure. This section discusses methods for improving the efficiency of Knuth-Bendix, making it fully automatic, augmenting it to allow for rewriting modulo a set of equations, extending it to handle first-order predicate calculus, and using it in alternative inductionless induction schemes.

6.2.4.1 Computing Small Critical Pairs

As discussed in Section 4.3.1, smaller critical pairs are more desirable than larger ones. It is difficult, if not impossible, to determine the size of a critical pair in advance (in general). However, Section 4.3.1 notes that it is a good heuristic to pick a small pair of untried rewrite rules with which to compute critical pairs.

Since critical pairs are expensive to compute, it is useful to generate only a few critical pairs at a time. If these can be ordered into rules, they might reduce or eliminate larger rules, reducing the number and size of other critical pairs that must be computed.

Section 4.3.1 noted that if we keep the rules sorted and always pick the smallest unmarked rule, the marking scheme will always use the smallest pairs of untried rules. A drawback, though, is that many critical pairs get generated at once.

[20] One such rewriting system, encountered in Bellegarde's work, is $\{f(g(x), g(y)) \longrightarrow g(f(x, y)),\ f(x, f(y, z)) \longrightarrow f(f(x, y), z),\ f(f(x, g(y)), g(z)) \longrightarrow f(x, g(f(y, z)))\}$.

We can generate fewer critical pairs at once if we pick the smallest *pair* of untried rules, and only compute the critical pairs between those two rules before attempting to order the critical pairs. This scheme requires more bookkeeping.

RRL extends this idea further, by generating critical pairs one at a time. Once the smallest pair of rewrite rules has been identified, only one critical pair (if any exist) is generated from the pair of rules. After handling the critical pair (e.g., by ordering it into a rewrite rule and normalizing the rewriting system accordingly), if that same pair of rules is still the smallest pair, the next critical pair between those rules is generated, and so on.

6.2.4.2 Fully-Automatic Knuth-Bendix

The Knuth-Bendix completion procedure, as implemented in REVE, does not yet work fully automatically. User interaction is required to:

(1) Choose one of the minimal extenders to try, whenever an equation is not orderable.

(2) Invoke the Knuth-Bendix "undo" command when (for some compatible equation) there are no further minimal extenders to try.

(3) Decide whether an incompatible equation should be divided, and, if so, what the name of the new operator should be.

(4) Interrupt Knuth-Bendix and invoke "undo" when it appears that the completion process is diverging, possibly because of some "bad" decision made earlier in the completion process.

Let us assume that we are using the automatic method for constructing a terminating rewriting system from a set of equations, as described in Section 3.6. This will automatically handle (1) and (2) above. Similarly, assume that REVE can automatically introduce a non-conflicting operator name when an incompatible equation is divided[21], so that (3) reduces to a decision of whether or not to divide the equation. In this context, a *decision path* for a system is a sequence of choices, one choice for each equation that requires a decision (choosing a minimal extender for a compatible equation or choosing whether or not to divide an incompatible equation) as Knuth-Bendix proceeds.

[21] Lescanne's REVE 1 provided this capability.

Knuth-Bendix could be fully automated by pursuing all possible decision paths for a system, to handle (3) and remove the need for (4). Recall that REVE uses a **system** data abstraction to store all state information required by Knuth-Bendix (see Section 5.4.5). At each decision point during the completion process, REVE could make duplicate copies of the current state of the **system**, one for each possible decision at that point, and put them into a *process queue*. REVE could then continue to complete each of the **systems** in the queue "simultaneously" by alternately running Knuth-Bendix for a short time on each of them. When any of these **systems** reaches another decision point, more **system** copies could be spawned, and so on. In effect, there would be one **system** in the process queue for each possible decision path. When any of the **systems** reaches a dead end (for some unorderable equation, there are no minimal extenders to try), that **system** would get deleted from the queue. Also, if criteria can be found that identify **systems** that are definitely diverging, such **systems** would also get deleted from the queue. If some decision path successfully produces a completed **system**, the process would stop. Since REVE would run Knuth-Bendix on each **system** in the process queue in an alternating fashion, the entire process would diverge only if all decision paths diverge.

As described here, it may appear that the fully-automatic Knuth-Bendix procedure would be hopelessly inefficient. However, as noted in Section 3.6, backtracking to choose different minimal extenders is usually not required. Also, most examples found in practice do not generate incompatible equations, so there are usually no decisions for dividing equations. Consequently, the speed of the scheme described above could probably be improved, in most cases, by giving running preference to the first **system**, and only pursuing other decision paths if the first **system** reaches a dead end, or requires an unusually long time to complete.

6.2.4.3 Equational Term Rewriting Systems

The correctness of Knuth-Bendix requires that the rewriting system terminate at each step of the procedure. As noted in Section 2.6, this requirement disallows the use of equation sets that include, e.g., the useful commutative equation $x + y = y + x$.

To handle this problem, Huet [Huet 80b] and Peterson & Stickel [Peterson 81] have extended the Knuth-Bendix procedure to operate on an *equational* term rewriting system (ETRS): a rewriting system, together with a set, E, of equations, where the equations in E are not converted into rules. For example, one might have E consist solely of the above commutative equation. The completed rewriting system, together with E, provides a decision procedure for the equational theory of the equations and rules that comprise the ETRS. Huet's method requires that all rewrite rules be left linear (for every rule, each variable appears at most once on the left-hand side). The Peterson-Stickel approach is limited to examples where E consists only of equations that are both left and right linear, and where a finite and complete unification algorithm for E is known. ("E-unification" is the process of finding a set of maximally-general substitutions for the variables in two terms, that make those two terms equal in the theory of E.)

ETRS completion procedures are powerful tools for automatic equational reasoning. Huet's procedure is too restrictive, however, to handle many typical examples of ETRS. The Peterson-Stickel procedure is probably too inefficient to permit a practical implementation. The inefficiency stems from both E-unification, wherein hundreds of substitutions are routinely computed for each pair of unifiable terms, and E-matching, wherein the equivalence class (under E) of a term is, in effect, searched to find an equivalent term that can be rewritten.

[Jouannaud 83] unifies the Huet and Peterson-Stickel results, showing them to be special cases of a more abstract theory. In addition, the [Jouannaud 83] approach generalizes Peterson-Stickel by allowing non-linear equations in E. However, [Jouannaud 83] does not propose a particular completion procedure that incorporates these new results.

In [Jouannaud 84], Jouannaud & Kirchner simplify, generalize, and extend the [Jouannaud 83] results about ETRS. They use these new results to prove the correctness of a new completion procedure that is more powerful and more efficient than previous methods. Some issues regarding the efficiency and effective use of the [Jouannaud 84] completion procedure are still under study. For automatic theorem proving applications, the failure-resistant properties described in Chapter 4 should also be considered for possible inclusion in the new ETRS completion procedure.

Helene Kirchner and Claude Kirchner are currently building on REVE 2 to create REVE 3, which will incorporate the [Jouannaud 84] completion procedure. Their implementation makes use of Yelick's generalized unification design and her implementation of a unification algorithm for AC theories [Yelick 84]. It is clear that the use of an ETRS completion procedure is essential for practical theorem proving using current rewriting methods.

6.2.4.4 First Order Predicate Calculus

Hsiang [Hsiang 82][22] has developed complete proof strategies for first order predicate calculus, based on rewriting methods and Knuth-Bendix. These strategies make use of a new, convergent ETRS for deciding Boolean algebra. Rather than using conventional, inefficient AC-unification for the Boolean binary operations, Hsiang introduced a new algorithm, called *BN-unification*, that is optimized for the Boolean operators. The validity of first order sentences is proven using a refutational proof technique that is much more efficient than resolution [Robinson 65] in many interesting cases. The utility of predicate calculus, and the efficiency of Hsiang's method, suggest that Hsiang's work should be included in a future release of REVE. In addition, it might be possible to use Hsiang's Boolean algebra ETRS to help perform the disjunctive normal form simplifications required by the COS minimal complete extender set computation scheme, described in Section 3.5.2.

6.2.4.5 Inductionless Induction

REVE uses the Huet-Hullot approach to inductionless induction [Huet 82], whose correctness requires that the rewriting system satisfy the principle of definition. However, as indicated in Section 2.7, this principle is undecidable in general, and REVE does not currently include a check for sufficient conditions. Jean-Jacques Thiel has recently proposed a powerful new algorithm for performing such a test [Thiel 84], and is currently implementing it in an experimental version of REVE.

The Huet-Hullot inductionless induction method is but one of several. Its principle advantage over other such methods is that it is fairly amenable to automatic theorem proving. Its principle disadvantage is that it disallows many interesting examples. Huet-Hullot requires that no two ground terms built from HH-constructors be congruent in the equational theory of the system. This restriction makes Huet-Hullot non-applicable to set theory, for example, since *insert* must be an HH-constructor (because *insert*(*empty*, *a*) is irreducible), and yet the theory of sets tells us that *insert*(*insert*(*empty*, *a*), *b*) and *insert*(*insert*(*empty*, *b*), *a*) are congruent.

[22] See also Hsiang & Dershowitz [Hsiang 83] for a condensed discussion of this work.

Kapur & Musser [Kapur 84b] have unified and generalized inductionless induction results into a general theory of *proof by consistency*. Their *unambiguity* property admits many interesting theories (including sets) that are not handled by Huet-Hullot. If useful (decidable) sufficient conditions can be identified that imply (undecidable) unambiguity, the Kapur-Musser approach may yield effective ways to handle many practical inductive theories in an automatic fashion.

[Huet 82] and [Lankford 81] discuss extensions, to ETRS, of their respective inductionless induction methods, where E is identically AC. Further work is needed to determine the applicability of inductionless induction and proof by consistency to more general E-theories.

6.2.5 Exploiting Concurrency Opportunities

The Knuth-Bendix completion procedure is inherently slow. Rewriting, ordering, unification, computing critical pairs, and "undo" backtracking are all fairly expensive operations. However, we remark that many of these functions are highly amenable to parallel processing:

- When rewriting a term, the left-hand sides of all rewrite rules can be simultaneously matched against the term. The rule corresponding to any successful match can be used to rewrite the term, since (in REVE) the order in which rules are applied does not matter.

- The efficiency of computing $s \succeq^{P} t$ (and hence $s \succeq^{*} t$) can be improved by comparing the subterms of s with the subterms of t in parallel. The particular subterms involved depend on the roots of s and t and the information in the registry. The efficiency of $s \succeq^{D} t$ can be similarly improved.

- There may be many critical pairs that result from overlapping the left-hand sides of two rewrite rules at all possible occurrences. All of these overlaps may be tried concurrently, since none of them depends on intermediate results from the other overlaps. In addition, multiple pairs of rules may be overlapped concurrently.

- The fully-automatic Knuth-Bendix implementation, described in Section 6.2.4.2, can be very time consuming, if multiple decision paths must be explored. The running time can be reduced by concurrently trying every decision path, rather than trying them in an alternating fashion.

Somewhat surprisingly, Dwork, Kanellakis, & Mitchell [Dwork 84] have shown that unification is an inherently sequential process that cannot benefit significantly from parallelism. However, they have also shown that matching, during rewriting, can be significantly improved using concurrency.

As concurrency capabilities in device technology, computer architecture, and programming languages increase, so will the potential speed and utility of automatic theorem proving methods using term rewriting techniques. The application of concurrency in this field is an interesting and largely unexplored research area.

6.3 Reflections on the System Development Process

Currently consisting of 20,000 lines of source code and in-line comments, more than four times the size of REVE 1, REVE 2 is one of the largest CLU programs in existence. It is only slightly smaller than the CLU compiler itself. Moreover, the size of REVE 2 is likely to grow by 50% in the next year, as the new features that will comprise REVE 3 get incorporated. In the presence of such a large and growing body of code, issues common to the development of all large software systems become almost as important as the application domain. In this section, we reflect on these issues as they pertain to REVE 2.

To maintain the consistency and coherence of the REVE source code as it evolves, full responsibility for maintaining REVE and incorporating improvements is always in the hands of a single person. Following an official REVE release by this maintainer, our colleagues are welcome to modify and extend the capabilities of REVE, using their own copy of the current source code. Before the next release, each such extension is sent to a small review committee for examination, to determine the importance of the extension and its degree of compatibility with the goals and existing code of the system. The selected extensions are combined, inconsistencies are resolved, and programming styles are made uniform, by the REVE maintainer. The new REVE version is released (with a new release number), and the development cycle repeats, building on the newly-released source code.

The CLU language provides a number of features that substantively assist in the construction of large programs. Data abstraction is fully supported in the language, and can be used to great effect in modularizing the code. Compile-time type checking of both built-in and user-defined types catches many errors that might otherwise result in obscure run-time bugs. Garbage collection, dynamic arrays, and exception handling automatically manage tedious and error-prone tasks and contribute to clean, elegant code. CLU's structured syntax is easy to read. Furthermore, the convenient CLU programming environment, with accompanying text editor and interactive symbolic debugger, expedites program development.

There are some disadvantages to using CLU, however. Most existing software in this field is written in LISP. LISP versions exist that have all of the benefits listed above for CLU, except for compile-time type checking and lucid syntax. Thus, it might appear that REVE's implementation language unnecessarily isolates REVE from similar efforts elsewhere. However, we believe that the benefits of compile-time type checking with large programs, even with the programming limitations that such checking imposes, outweigh the disadvantages of not using LISP. Moreover, even LISP code is not necessarily easy to share, because of differences in LISP dialects and installation environments.

The REVE implementation effort has illustrated an important tenet of large experimental systems development: Build a prototype early. A prototype implementation gives focus to the project, and helps indicate the potential difficulties to be tackled. Lescanne's REVE 1 served as an excellent prototype for REVE 2 by defining the problem area and providing preliminary solutions. However, this principle was not strictly followed during the development of REVE 2. We sometimes strove unnecessarily to offset potential efficiency difficulties, before it was established that such difficulties existed. For example, it was believed early on that the code could be made more efficient by keeping, in each operator appearing in a term, a "pointer" to the current registry. In this way, the registry would not have to be passed through multiple layers of procedure calls[23] to be available to the orderings. However, as this idea permeated the code, and every operator in every term that got copied or created had to contain the current registry, the incurred overhead cancelled any efficiency advantage. Moreover, the code became more convoluted and terms occupied far more storage than they needed to. In retrospect, it is probable that REVE 2 would have achieved its current features and performance more quickly if a simple implementation were built first, and selected modules were later replaced to effect efficiency or functionality improvements.

Clean and well-commented source code, identification of generally-useful abstractions, and careful module testing were found to be indispensable in the development of REVE 2. Implementation proceeded bottom-up[24], and the documentation, generality, and reliability of lower-level modules facilitated the coding of higher-level modules, allowing bugs to be quickly

---

[23]It is not desirable to maintain a global variable containing the current registry, because this precludes maintaining multiple systems, each with its own registry, in future versions of REVE.

[24]The design, however, of REVE 2 was performed top-down.

identified and removed. To date, no bugs have turned up in the basic data abstractions — terms, multisets, graphs, etc. — since they passed their original module tests at the time they were developed.

Several people at MIT have successfully written programs that use some or all of the modules in REVE 2. Yelick has used many of REVE's lower-level modules in her implementation of associative-commutative (AC) unification [Yelick 84]. Joseph Zachary used most of REVE in his system to experiment with permutative rewrite rules. Ronald Richards used parts of REVE in a program to compute overlap closures [Guttag 83a] for proving restricted termination of rewriting systems. David Detlefs completely re-engineered REVE's user interface, implemented EPOS and its minimal complete extender set algorithm, and assumed maintenance responsibility for REVE, without difficulty. In addition, Kownacki [Kownacki 84] has studied REVE's implementation, and has shown how REVE can be used for the theorem proving required to perform many of the semantic checks used in the Larch specification language [Guttag 83b]. REVE will be used as an embedded theorem prover in a support system for Larch, which includes a structured editor [Zachary 83] and semantic checker.

REVE has also been used in implementation work at other universities. Researchers at CRIN, the State University of New York at Stony Brook, and the University of Illinois at Urbana-Champaign are extending REVE and embedding parts of REVE in other applications.

Appendix:

REVE Commands

In this Appendix, we present the descriptions of each command in the current version of REVE 2. These descriptions are taken almost directly from the on-line HELP information provided by REVE. The commands fall into five categories, indicated by the subheadings below.

You do not need to type in the whole command name; unambiguous prefixes are sufficient. Commands and arguments can be typed in upper and/or lower case. If you have a file in your login directory called ".reve_init," that file will be executed as if by the REPLAY command whenever you start REVE. This can be useful, for example, if you often desire a page mode, tracing level, etc., that is different from the default, or if you always want to script your sessions.

User Interaction

HELP	Provides the user with detailed explanations of REVE commands, as well as information on other topics, such as interrupting Knuth-Bendix, or entering arguments to commands. HELP takes one argument, which is the topic on which help is desired. Unambiguous prefixes are sufficient specifiers of help topics. "HELP ?" prints out a terse list of topics on which help is available. "GENERAL" is a special topic that gives a short introduction to each HELP topic.
TRACE	Sets the Knuth-Bendix tracing level. This should be an integer between 0 and 3, inclusive. 0 is the least verbose, printing nothing but user interaction. Level 1 announces the size of the system at regular intervals, and informs the user whether Knuth-Bendix is reducing and orienting equations or generating critical pairs, that equations have been oriented into rewrite rules, that rewrite rules have been turned back into equations because their left-hand sides were reduced, that non-trivial critical pairs have been found, and that equations have been divided or separated. Level 2 gives this information, and also informs the user when an equation or the right hand side of a rewrite rule has been reduced as a result of the addition of a new rule, and, for critical pairs, gives both the original critical pair and its reduced form. Finally, level 3 gives all this information, and

	also informs the user when equations are postponed because they are classified as "big" or are unable to be ordered, and also always prints the pair of rules being superposed, even if no critical pairs are found. 1 is the default tracing level. An argument of "?" displays the current tracing level.
SCRIPT	Starts recording of the terminal session in a script file. SCRIPT takes an argument, which is the name of the file to which scripting should be sent. Any previous contents of a script file are lost. Only one script file is allowed at a time. Scripting is ended by the QUIT or UNSCRIPT commands.
UNSCRIPT	Stops recording the terminal session in a script file, and closes that file.
LOG	Starts recording the user input in a log file. LOG takes an argument, which is the name of the file to which logging should be sent. Any previous contents of the file are lost. Only one log file is allowed at a time. Logging is ended by the QUIT or UNLOG commands. In order to avoid annoying UNLOG commands at the end of log files, UNLOG commands are not stored in log files. Log files, once made, can be executed via the REPLAY command. (REPLAY commands are not stored in log files, either.)
UNLOG	Stops the recording of user input in a log file. The log file is closed. UNLOG commands do not show up in log files.
REPLAY	Causes REVE to take input from the file whose name is given as the argument. This command is ordinarily used to read from a file that was created by the LOG command, but any text file may be specified. Once the file has been exhausted, REVE starts accepting input from the terminal. REPLAY commands may not be nested, so REPLAY commands are ignored in files executed via the REPLAY command. REPLAY commands do not appear in log files.
PAGE	Controls REVE's page mode. In page mode, REVE buffers output a screen at a time, so that no output is missed. When a screenful of output has been printed since the last user interaction, the user is prompted for what to do next. The options include printing the next full screen, half screen, single line, or "n" lines where n is a single digit; printing without stopping until the next user interaction point; or not printing at all until the next user interaction point. These options are explained in detail if you type "?" in response to a "--More--" prompt. The default page mode is "off." See HELP REVE-INIT for information on how to change this. (Another method of controlling output is by using the ↑S and ↑Q keys. ↑S stops output, and ↑Q resumes printing.)
QUIT	Causes REVE to halt, returning the user to the operating system. Any script or log file is closed.

Input/Output

READ	Deletes any existing equations and rewrite rules in REVE, and reads new equations from a file. The precedence information is cleared, and the status of all operators becomes "undefined." The file name is given as an argument. If the file name has no directory part, the current working directory is first searched for that file, and then a special "examples" directory is searched. An argument of "?" gives a list of the example equation files in this directory. See also the TERMINAL, APPEND, and ADDITIONAL commands.
APPEND	Reads equations from a file, adding them to the current system. The file name is given as an argument. If the file name has no directory part, the current working directory is first searched for that file, and then a special "examples" directory is searched. An argument of "?" gives a list of the example equation files in this directory. See also the READ, TERMINAL, and ADDITIONAL commands.
TERMINAL	Deletes any existing equations and rewrite rules in the system, and reads new equations from the terminal. The precedence information is cleared, and the status of all operators becomes "undefined." See also the READ, APPEND, and ADDITIONAL commands.
ADDITIONAL	Reads new equations from the terminal, and adds them as user equations to the system. See also the READ, TERMINAL, and APPEND commands.
WRITE	Writes the equations and rewrite rules in the current system to a file, given as the argument. This file can later be read in (with the rewrite rules interpreted as equations) using the READ or APPEND commands.
DISPLAY	Displays the equations and rewrite rules in the current system on the terminal. Divides the equations into two sets, those entered by the user and those generated as critical pairs. The equations and rules are numbered for reference in other commands. Also shows the equation to be proved if an equational or inductive proof is in progress.
FREEZE	Saves the current system, including the equations, rewrite rules, precedence, status map, and Knuth-Bendix "undo" information, into a file in object form. The name of this file is given as the argument. Systems saved using FREEZE can be later be restored by the THAW command. This command is useful for saving completed or partially-completed systems.
THAW	Restores a system that was saved previously using the FREEZE command. The name of the file in which the system was saved is given as the argument. THAW does not allow files to be thawed if they were made using an out-of-date version of REVE.

System

KB Runs the Knuth-Bendix Completion Procedure on the current system of equations and rewrite rules. Knuth-Bendix attempts to complete a system into a convergent rewriting system with the same equational theory as the original set of user equations. This rewriting system can be used to prove theorems using the PROVE command, or to reduce terms to their normal form using the NORMAL-FORM command. The TRACE command controls output produced by Knuth-Bendix during the completion process. You can interrupt Knuth-Bendix with ↑G, and resume it with KB.

UNDO Causes the system to be set to its state before the last interaction with Knuth-Bendix, if there was one, and restarts Knuth-Bendix from that point.

PROVE Attempts to prove that an equation is a theorem with the respect to the equations and rewrite rules in the system. The equation is given as the argument. This command first attempts to prove the equation by rewriting, even if the system is only partially completed. If the current system is not yet completed, this command then asks the user if he wants to run Knuth-Bendix to get a convergent set of rewrite rules, and does so if he answers yes. PROVE then attempts to prove the theorem by normalizing both sides of the equation. If they have the same normal form, the equation is a theorem in the equational theory. Otherwise, this command then asks if the user wants to attempt to prove that the equation is an inductive theorem. If the answer is "yes," PROVE attempts to use inductionless induction to prove the theorem. This involves adding the equation to be proved to the system, and then running Knuth-Bendix again. If Knuth-Bendix terminates successfully, the equation is in the inductive theory. If it terminates with "disproof," the equation is not valid. If Knuth-Bendix does not terminate, it is not known whether the equation is a theorem. In order for inductionless induction to work properly, the user should declare Huet-Hullot constructors using the HH-CONSTRUCTORS command (to use the Huet-Hullot method), or should fully axiomatize equality and declare "true" and "false" as hh-constructors (to use Musser's method.) The user may interrupt Knuth-Bendix during PROVE; the system will remember what equation was being proved. Use the KB command to continue an interrupted inductive proof. The user may also cancel a proof using the CANCEL command.

CANCEL Cancels any proof currently in progress. To determine if there is a proof in progress, use the DISPLAY command. See PROVE for more information on proofs.

DELETE Takes a list of integers, separated by spaces or tabs, as its argument. These integers should correspond to the numbers associated with equations and rewrite rules as shown in the DISPLAY command. (DELETE, followed by <RETURN>, causes a DISPLAY to be performed before the

	user is prompted for the equations and rules to delete.) If all the numbers in the list correspond to equations and/or rules in the system, those equations and rules are deleted. Otherwise, nothing is done, and an error message is printed. Deleting rewrite rules "compromises" the system, in that it is no longer guaranteed to represent the same equational theory as the original system. "Deleted" critical pair equations are saved on a special list, and are reinserted into the system after Knuth-Bendix is finished, to preserve correctness. In this case, DELETE should be thought of as postponing consideration of an equation. Deleting user equations just causes Knuth-Bendix to complete the system consisting of the new, smaller set of equations.
CLEAR	Resets REVE. All equations and rewrite rules are deleted from the system, the precedence is cleared, and the status of all operators is set to "undefined."
TASK-ORDER	Changes the order in which the Knuth-Bendix tasks are executed. The default is "automatic," a task order that considers all non-big unorderable equations before computing critical pairs. This order is the most efficient one for use with automatic orderings, such as EPOS, and perhaps also for the current implementation of EDOS, if you are familiar with the basics of choosing EDOS suggestions. The other possible task order currently available is "postpone," which postpones compatible unorderable equations until after critical pairs have been computed, in the hope that the unorderable equations will reduce (and become orderable) or become identities and disappear. If you interrupt Knuth-Bendix and change the task ordering, Knuth-Bendix will start with the first task of the new order when resumed.
AUTOMATIC	Sets the current REVE execution mode to be automatic ("on") or manual ("off"). If "on," and the current ordering is EPOS, the ORIENT command will convert the equations into rewrite rules without user help, automatically choosing different minimal extenders, reversing equations when all extenders have been tried, etc. In the future, if a fully-automatic Knuth-Bendix is implemented, AUTOMATIC will also determine whether or not Knuth-Bendix runs automatically.
ORIENT	Causes REVE to order all current equations into rewrite rules, using the current ordering, without computing any critical pairs. If the AUTOMATIC execution mode is "on," and the current ordering is EPOS, ORIENT will transform the equations into rules without user help, automatically choosing different minimal extenders, reversing equations when all extenders have been tried, and reporting failure if a terminating registry cannot be found. Otherwise, for each unorderable equation, the suggestions or extenders from the ordering are displayed, and the user is prompted to take action on the equation accordingly.

Laboratory

REDUCE	Rewrites a term once, using the current rewriting system. The term is given as the argument. The choice of rewrite rule applied is non-deterministic.
NORMAL-FORM	Computes the normal form of a term with respect to the current rewriting system. The term is given as the argument. If the rewriting system is not guaranteed to terminate, i.e., if the user has added a rewrite rule to the system that the current ordering was unable to order (or if the "manual" ordering is being used), the normal form computation may not terminate. When the rewriting system is not known to terminate, REVE stops the rewriting process and issues a warning after a very large number of rewrites during a normal form computation.
UNIFY	Computes and prints the unification of two terms, i.e., the result of applying their most general unifier to either term. Standard unification (i.e., unification in the empty theory) is used. The two terms are entered as arguments. If the terms are entered on the same line, they should be separated by a semicolon (";").
CRITICAL-PAIRS	Finds and prints all critical pairs between two rewrite rules, which are entered as arguments. If the two rules are entered on the same line, they should be separated by a semicolon (";").

Orderings

ORDERING	Sets the ordering to be used by Knuth-Bendix. Currently, the orderings supported are "EPOS," which computes the minimal complete extender set when an equation is unorderable; "EDOS," which currently provides suggestions for extending the ">" relation in the precedence, and "manual," which prompts the user to hand-order each equation. When ORDERING is used to switch from "manual" to either "EPOS" or "EDOS," all rewrite rules are converted back into equations to preserve the correctness of Knuth-Bendix.
INITIALIZE	Restores the system to a state in which there are no Huet & Hullot constructors, there is no precedence information associated with operators, and all operators have "undefined" status. All rewrite rules are turned back into equations. Note that this preserves the equational theory defined by the rewrite rules and equations in the system. See also the CLEAR command.
PRECEDENCE	Adds precedence information to the system. REVE uses orderings on terms to prove the termination of the rewriting system. These orderings are parameterized on a precedence. The precedence records information regarding whether "f > g," "f = g," "f >= g," or "f" and "g" are unre-

lated, with respect to the ordering, for any two operators "f" and "g." PRECEDENCE takes an argument, which is a list of lists of relations among operators, where the lists are separated by commas. The permissible relations are ">," "<," "=," ">=," and "<=." For example, the argument "f >= g < h, a = g" causes "f" to be greater than or equal to "g," "h" to be greater than "g," and "a" and "g" to be equivalent in the precedence. All operators in the lists must already appear in one of the equations or rewrite rules in the system. All the lists taken together must parse correctly and represent a consistent addition to the precedence, or else nothing is done and an error message is printed. See also the CONSTRUCTORS, STATUS, and OPERATORS commands.

STATUS Declares the status of an operator, which is used by the orderings in REVE. This status can be "multiset," "left-to-right," "right-to-left," or "undefined." Loosely, "multiset" status for "f" means that for a term "t" = "f(...)," the ordering regards the arguments of "t" as a multiset, and the order of the arguments is ignored. When the status is "left-to-right," the leftmost arguments of "t" are given more weight in the ordering. Similarly, "right-to-left" status indicates that the rightmost arguments are more important. If the status of "f" is "undefined," "f" has not yet been assigned a particular status. "Undefined" is the initial status assignment of all operators. STATUS takes two arguments: an operator name, and a status, which should be "left" for left-to-right, "right" for right-to-left, or "multiset." The operator must already appear in one of the equations or rewrite rules in the system. See also the PRECEDENCE, CONSTRUCTORS, and OPERATORS commands.

CONSTRUCTORS Adds precedence information to the system. REVE uses an ordering on terms to prove the termination of the rewriting system. This ordering on terms is an extension of a partial ordering on operators, called a precedence. CONSTRUCTORS allows one to extend the precedence relation in a particular way: It takes one argument, which is a list of operators, and declares each of those operators to be less than or equal to all other operators not in the list. You must declare all constructors at the same time. It is particularly useful to declare all of the basic constructors using this command (hence its name), since all constructors are almost always less than all non-constructors in any precedence that allows REVE's ordering to prove termination. All operators declared using this command must already appear in one of the equations or rewrite rules in the system. See also the PRECEDENCE, STATUS, OPERATORS, and HH-CONSTRUCTORS commands.

HH-CONSTRUCTORS
Declares Huet-Hullot constructors, which are used in inductionless induction. For inductionless induction to work properly, it must be the case that every "ground term" (a term containing no variables) is congruent, with respect to the equations and rewrite rules in the system, to exactly one

ground term consisting solely of Huet-Hullot constructors. In abstract data type axioms, the constructors of the data type will often have this property. (Sets are a notable exception, since "insert" and "new" are the set constructors, and, in general, there will be many congruent ground terms that denote a given set.) This command takes one argument, which is a list of operators. Any operators that get declared as Huet-Hullot constructors also receive the treatment accorded to operators by the CONSTRUCTORS command. All operators declared using HH-CONSTRUCTORS must already appear in one of the equations or rewrite rules in the system. See also the CONSTRUCTORS command.

OPERATORS Displays the operator precedence and status information in the system. For every set of operators that are equivalent in the precedence, tells which of those operators are constructors ("(C)"), and/or have non-undefined status ("(M)" for "multiset," "(L)" for "left-to-right," or "(R)" for "right-to-left"), and displays the operators to which they are greater than or equal in the precedence. OPERATORS without any arguments prints this information for all the operators in the system. If a list of operators is typed on the same line as the OPERATORS command, only information about the relationships between these operators is listed.

CHECK Checks the operator information for certain kinds of inconsistencies. It will tell the user if there are any sets of equivalent operators that contain both constructors and non-constructors, or operators of both "multiset" and lexicographic ("left-to-right" or "right-to-left") status. This situation must be corrected before Knuth-Bendix can be run, so REVE automatically performs CHECK before running Knuth-Bendix. CHECK is not guaranteed to catch all such inconsistencies in a single pass; only to catch at least one if there are any.

References

[Aho 72] A. V. Aho, M. R. Garey, and J. D. Ullman, "The Transitive Reduction of a Directed Graph," *SIAM Journal of Computing* 1(2):131-137, June 1972.

[Backus 78] J. Backus, "Can Programming Be Liberated from the von Neumann Style? A Functional Style and its Algebra of Programs," *Communications of the ACM* 21(8):613-641, August 1978.

[Barros 84] L. Barros and J. L. Remy, "ECOLOGISTE, A System to Make Complete and Consistent Specifications Easier," *Proc. of an NSF Workshop on the Rewrite Rule Laboratory, Sept. 6-9, 1983*, General Electric Corporate Research and Development Report No. 84GEN008, Schenectady, NY, April 1984, pp. pp. 301-318.

[Baxter 73] L. D. Baxter, "An Efficient Unification Algorithm," Technical Report CS-73-23, Dept. of Applied Analysis and Computer Science, Univ. of Waterloo, Waterloo, Ontario, 1973.

[Bellegarde 84] F. Bellegarde, "Rewriting Systems on FP Expressions that Reduce the Number of Sequences They Yield," *Proc. 1984 ACM Conf. on LISP and Functional Programming*, Austin, TX, August 1984.

[Butler 80] G. Butler and D. S. Lankford, "Experiments with Computer Implementations of Procedures Which Often Derive Decision Algorithms for the Word Problem in Abstract Algebras," Technical Report MTP-7, Louisiana Tech. Univ., 1980.

[Cohen 69] P. J. Cohen, "Decision Procedures for Real and *p*-adic Fields," *Comm. Pure Appl. Math.* 22(2):131-151, 1969.

[Corbin 83] J. Corbin and M. Bidoit, "A Rehabilitation of Robinson's Unification Algorithm," R. E. A. Mason (Ed.), *Proc. 9th World Computer Congress, IFIP '83*, North-Holland, September 1983, pp. 909-914.

[Dershowitz 79a] N. Dershowitz and Z. Manna, "Proving Termination with Multiset Orderings," *Communications of the ACM* 22(8):465-476, 1979. Preliminary version in *Proc. 6th EATCS Intl. Colloq. on Automata, Languages, and Programming*, Graz, Austria, July 1979, pp. 188-202.

[Dershowitz 79b] N. Dershowitz, "A Note on Simplification Orderings," *Information Processing Letters* 9(5):212-215, 1979.

[Dershowitz 82a] N. Dershowitz, "Orderings for Term-Rewriting Systems," in *Theoretical Computer Science, Vol. 17*, North-Holland, 1982, pp. 279-301. Preliminary version in *Proc. 20th IEEE Symp. on Foundations of Computer Science*, San Juan, Puerto Rico, October 1979, pp. 123-131.

[Dershowitz 82b] N. Dershowitz, "Existence and Construction of Rewrite Systems," Technical Report ATR-82(8478)-3, Aerospace Corp., El Segundo, CA, December 1982.

[Dershowitz 83a] N. Dershowitz, "Computing with Rewrite Systems," Technical Report ATR-83(8478)-1, Aerospace Corp., El Segundo, CA, January 1983. Also in *Proc. of an NSF Workshop on the Rewrite Rule Laboratory, Sept. 6-9, 1983*, General Electric Corporate Research and Development Report No. 84GEN008, Schenectady, NY, April 1984, pp. 269-298. To appear in *Information and Control*.

[Dershowitz 83b] N. Dershowitz, "Applications of the Knuth-Bendix Completion Procedure," Technical Report ATR-83(8478)-2, Aerospace Corp., El Segundo, CA, May 1983.

[Dershowitz 83c] N. Dershowitz, "Well-Founded Orderings," Technical Report ATR-83(8478)-3, Aerospace Corp., El Segundo, CA, May 1983.

[Dwork 84] C. Dwork, P. C. Kanellakis, and J. C. Mitchell, "On the Sequential Nature of Unification," *Journal of Logic Programming* 1(1), 1984.

[Evans 67] T. Evans, "Products of Points — Some Simple Algebras and Their Identities," *Amer. Math. Monthly* 74, pp. 362-372, 1967.

[Fay 79] M. Fay, "First-order Unification in an Equational Theory," *Proc. 4th Workshop on Automated Deduction*, Austin, TX, February 1979, pp. 161-167. Also Master's Thesis, Technical Report 78-5-002, Univ. of California, Santa Cruz, May 1978.

[Forgaard 84] R. Forgaard and J. V. Guttag, "REVE: A Term Rewriting System Generator with Failure-Resistant Knuth-Bendix," *Proc. of an NSF Workshop on the Rewrite Rule Laboratory, Sept. 6-9, 1983*, General Electric Corporate Research and Development Report No. 84GEN008, Schenectady, NY, April 1984, pp. pp. 5-31.

[Göbel 84] R. Göbel, "A Completion Procedure for Globally Finite Term Rewriting Systems," *Proc. of an NSF Workshop on the Rewrite Rule Laboratory, Sept. 6-9, 1983*, General Electric Corporate Research and Development Report No. 84GEN008, Schenectady, NY, April 1984, pp. 155-203.

[Goguen 79] J. A. Goguen and J. J. Tardo, "An Introduction to OBJ: A Language for Writing and Testing Formal Algebraic Program Specifications," *Proc. Specification of Reliable Software*, Institute of Electrical and Electronics Engineers, April 1979, pp. 170-189.

[Goguen 80] J. A. Goguen, "How to Prove Algebraic Inductive Hypotheses Without Induction, With Applications to the Correctness of Data Type Implementation," *Lecture Notes in Computer Science, Vol. 87: Proc. 5th Conf. on Automated Deduction*, Les Arcs, France, Springer-Verlag, New York, July 1980, pp. 356-373.

[Goree 81] J. A. Goree, Jr., "Using Abstractions to Implement the Knuth-Bendix Completion Procedure," Bachelor's Thesis, MIT Lab. for Computer Science, May 1981.

[Guttag 78a] J. V. Guttag and J. J. Horning, "The Algebraic Specification of Abstract Data Types," *Acta Informatica* 10, pp. 27-52, 1978.

[Guttag 78b] J. V. Guttag, E. Horowitz, and D. R. Musser, "Abstract Data Types and Software Validation," *Communications of the ACM* 21(12):1048-1064, December 1978.

[Guttag 83a] J. V. Guttag, D. Kapur, and D. R. Musser, "On Proving Uniform Termination and Restricted Termination of Rewriting Systems," *SIAM Journal of Computing* 12(1):189-214, February 1983. Also as "Derived Pairs, Overlap Closures, and Rewrite Dominoes: New Tools for Analyzing Term Rewriting Systems," *Lecture Notes in Computer Science, Vol. 40: Proc. 9th EATCS Intl. Colloq. on Automata, Languages, and Programming*, Aarhus, Denmark, July 1982, pp. 300-312, and Technical Report TR-268, MIT Lab. for Computer Science, December 1981.

[Guttag 83b] J. V. Guttag and J. J. Horning, "Preliminary Report on The Larch Shared Language," Technical Report TR-307, MIT Lab. for Computer Science, October 1983. Also Technical Report CSL-83-6, Xerox Palo Alto Research Center, Palo Alto, CA, September 1983.

[Hsiang 82] J. Hsiang, "Topics in Automated Theorem Proving and Program Generation," Ph.D. Thesis, Univ. of Illinois, Urbana-Champaign, November 1982.

[Hsiang 83] J. Hsiang and N. Dershowitz, "Rewrite Methods for Clausal and Non-Clausal Theorem Proving," *Lecture Notes in Computer Science, Vol. 154: Proc. 10th EATCS Intl. Colloq. on Automata, Languages, and Programming*, Barcelona, Spain, Springer-Verlag, New York, July 1983, pp. 331-346.

[Huet 78] G. Huet and D. S. Lankford, "On the Uniform Halting Problem for Term Rewriting Systems," Laboratory Report 283, INRIA, Le Chesnay, France, March 1978.

[Huet 80a] G. Huet and D. C. Oppen, "Equations and Rewrite Rules: A Survey," in R. Book (Ed.), *Formal Language Theory: Perspectives and Open Problems*, Academic Press, New York, 1980, pp. 349-405.

[Huet 80b] G. Huet, "Confluent Reductions: Abstract Properties and Applications to Term Rewriting Systems," *Journal of the ACM* 27(4):797-821, October 1980. Preliminary version in *Proc. 18th IEEE Symp. on Foundations of Computer Science*, Providence, RI, October 1977, pp. 30-45.

[Huet 81] G. Huet, "A Complete Proof of Correctness of the Knuth-Bendix Completion Algorithm," *Journal of Computer and System Sciences* 23(1):11-21, August 1981.

[Huet 82] G. Huet and J.-M. Hullot, "Proofs by Induction in Equational Theories with Constructors," *Journal of the ACM* 25, pp. 239-266, 1982. Preliminary version in *Proc. 21st IEEE Symp. on Foundations of Computer Science*, Los Angeles, CA, October 1980, pp. 96-107.

[Hullot 80a] J.-M. Hullot, "A Catalogue of Canonical Term Rewriting Systems," Technical Report CSL-113, SRI Intl. Computer Science Laboratory, Menlo Park, CA, April 1980.

[Hullot 80b] J.-M. Hullot, "Canonical Forms and Unification," *Lecture Notes in Computer Science, Vol. 87: Proc. 5th Conf. on Automated Deduction*, Les Arcs, France, Springer-Verlag, New York, July 1980, pp. 318-334.

[Iturriaga 67] R. Iturriaga, "Contributions to Mechanical Mathematics," Ph.D. Thesis, Carnegie-Mellon Univ., Pittsburgh, PA, 1967.

[Jouannaud 82a] J.-P. Jouannaud, P. Lescanne, and F. Reinig, "Recursive Decomposition Ordering," *Proc. 2nd IFIP Workshop on Formal Description of Programming Concepts*, Garmisch-Partenkirchen, W. Germany, June 1982. Also in "Recursive Decomposition Ordering and Multiset Orderings," Technical Memo TM-219, MIT Lab. for Computer Science, June 1982.

[Jouannaud 82b] J.-P. Jouannaud and P. Lescanne, "On Multiset Ordering," *Information Processing Letters* 15(2), 1982. Also in J.-P. Jouannaud, P. Lescanne, and F. Reinig, "Recursive Decomposition Ordering and Multiset Orderings," Technical Memo TM-219, MIT Lab. for Computer Science, June 1982.

[Jouannaud 83] J.-P. Jouannaud, "Church-Rosser Computations with Equational Term Rewriting Systems," Technical Report, Centre de Recherche en Informatique de Nancy, Vandoeuvre-lès-Nancy, France, January 1983. Submitted to *Journal of the ACM*.

[Jouannaud 84] J.-P. Jouannaud and H. Kirchner, "Completion of a Set of Rules Modulo a Set of Equations," Technical Note, SRI Intl. Computer Science Laboratory, Menlo Park, CA, April 1984. Preliminary version in *Proc. 11th ACM Symp. on Principles of Programming Languages*, Salt Lake City, UT, January 1984. Also in *Proc. of an NSF Workshop on the Rewrite Rule Laboratory, Sept. 6-9, 1983*, General Electric Corporate Research and Development Report No. 84GEN008, Schenectady, NY, April 1984, pp. 207-228. Submitted to *SIAM Journal of Computing*.

[Kamin 80] S. Kamin and J.-J. Lévy, "Attempts for Generalising the Recursive Path Orderings," Dept. of Computer Science, Univ. of Illinois, Urbana-Champaign, February 1980. Unpublished manuscript.

[Kapur 84a] D. Kapur and G. Sivakumar, "Experiments with and Architecture of RRL, A Rewrite Rule Laboratory," *Proc. of an NSF Workshop on the Rewrite Rule Laboratory, Sept. 6-9, 1983*, General Electric Corporate Research and Development Report No. 84GEN008, Schenectady, NY, April 1984, pp. 33-56.

[Kapur 84b] D. Kapur and D. R. Musser, "Proof by Consistency," Report GECRD-84-083, General Electric Corporate Research and Development, Schenectady, NY, February 1984. Also in *Proc. of an NSF Workshop on the Rewrite Rule Laboratory, Sept. 6-9, 1983*, General Electric Corporate Research and Development Report No. 84GEN008, Schenectady, NY, April 1984, pp. 245-267.

[Knuth 70] D. E. Knuth and P. B. Bendix, "Simple Word Problems in Universal Algebras," in J. Leech (Ed.), *Computational Problems in Abstract Algebra*, Pergamon, Oxford, 1970, pp. 263-297.

[Kowalski 74] R. A. Kowalski, "Predicate Logic as a Programming Language," *Proc. IFIP-74 Congress*, North-Holland, 1974, pp. 569-574.

[Kownacki 84] R. W. Kownacki, "Semantic Checking of Formal Specifications," Master's Thesis, MIT Lab. for Computer Science, June 1984.

[Lankford 75a] D. S. Lankford, "Canonical Algebraic Simplification in Computational Logic," Technical Report ATP-25, Automatic Theorem Proving Project, Univ. of Texas, Austin, May 1975.

[Lankford 75b] D. S. Lankford, "Canonical Inference," Technical Report ATP-32, Mathematics Dept., Univ. of Texas, Austin, December 1975.

[Lankford 79a] D. S. Lankford, "On Proving Term Rewriting Systems are Noetherian," Technical Report MTP-3, Mathematics Dept., Louisiana Tech. Univ., May 1979.

[Lankford 79b] D. S. Lankford and A. M. Ballantyne, "The Refutation Completeness of Blocked Permutative Narrowing and Resolution," *Proc. 4th Workshop on Automated Deduction*, Austin, TX, February 1979, pp. 53-59.

[Lankford 81] D. S. Lankford, "A Simple Explanation of Inductionless Induction," Technical Report MTP-14, Mathematics Dept., Louisiana Tech. Univ., August 1981.

[Lescanne 83a] P. Lescanne, "Computer Experiments with the REVE Term Rewriting System Generator," *Proc. 10th ACM Symp. on Principles of Programming Languages*, Austin, TX, January 1983, pp. 99-108.

[Lescanne 83b] P. Lescanne, "Some Properties of Decomposition Ordering. A Simplification Ordering to Prove Termination of Rewriting Systems," *RAIRO Informatique Theorique/Theoretical Informatics* 16, pp. 331-347, 1983.

[Lescanne 84] P. Lescanne, "Uniform Termination of Term Rewriting Systems: Recursive Decomposition Ordering with Status," *Proc. of 6th Colloq. on Trees in Algebra and Programming*, Bordeaux, France, Cambridge Univ. Press, March 1984. Also as "How to Prove Termination? An Approach to the Implementation of a new Recursive Decomposition Ordering," *Proc. of an NSF Workshop on the Rewrite Rule Laboratory, Sept. 6-9, 1983*, General Electric Corporate Research and Development Report No. 84GEN008, Schenectady, NY, April 1984, pp. 109-121.

[Lipton 77] R. J. Lipton and L. Snyder, "On the Halting of Tree Replacement Systems," *Proc. Conf. on Theoretical Computer Science*, Univ. of Waterloo, Waterloo, Ontario, August 1977, pp. 43-46.

[Liskov 81] B. Liskov, R. Atkinson, T. Bloom, E. Moss, J. C. Schaffert, R. Scheifler, A. Snyder, *Lecture Notes in Computer Science, Vol. 114: CLU Reference Manual*, Springer-Verlag, New York, 1981.

[Manna 70] Z. Manna and S. Ness, "On the Termination of Markov Algorithms," *Proc. 3rd Hawaii Intl. Conf. on System Sciences*, Honolulu, HI, January 1970, pp. 789-792.

[Martelli 82] A. Martelli and U. Montanari, "An Efficient Unification Algorithm," *ACM Transactions on Programming Languages and Systems* 4(2):258-282, April 1982.

[Musser 80a] D. R. Musser, "Abstract Data Type Specification in the Affirm System," *IEEE Transactions on Software Engineering* 6(1):24-32, January 1980.

[Musser 80b] D. R. Musser, "On Proving Inductive Properties of Abstract Data Types," *Proc. 7th ACM Symp. on Principles of Programming Languages*, Las Vegas, NV, January 1980, pp. 154-162.

[Musser 84] D. R. Musser, "The L Programming Language Preliminary Reference Manual," *Proc. of an NSF Workshop on the Rewrite Rule Laboratory, Sept. 6-9, 1983*, General Electric Corporate Research and Development Report No. 84GEN008, Schenectady, NY, April 1984, pp. 57-72.

[Newman 42] M. H. A. Newman, "On Theories With a Combinatorial Definition of 'Equivalence'," *Annals of Mathematics* 43(2):223-243, 1942.

[Paterson 78] M. S. Paterson and M. N. Wegman, "Linear Unification," *Journal of Computer and System Sciences* 16, pp. 158-167, 1978.

[Peterson 81] G. E. Peterson and M. E. Stickel, "Complete Sets of Reductions for Some Equational Theories," *Journal of the ACM* 28(2):233-264, April 1981.

[Plaisted 78a] D. A. Plaisted, "Well-Founded Orderings for Proving Termination of Systems of Rewrite Rules," Report R-78-932, Dept. of Computer Science, Univ. of Illinois, Urbana-Champaign, July 1978.

[Plaisted 78b] D. A. Plaisted, "A Recursively Defined Ordering for Proving Termination of Term Rewriting Systems," Report R-78-943, Dept. of Computer Science, Univ. of Illinois, Urbana-Champaign, September 1978.

[Plaisted 83] D. A. Plaisted, private communication, September 1983.

[Remy 84] J. L. Remy and H. Zhang, "REVEUR 4: A System for Validating Conditional Algebraic Specifications of Parameterized Abstract Data Types," *Proc. 2nd European Conference on Artificial Intelligence*, Pisa, Italy, September 1984. (To appear.)

[Remy 85] J. L. Remy and H. Zhang, "REVEUR 4: A System to proceed Experiments on Conditional Term Rewriting Systems," January 1985. Unpublished manuscript.

[Robinson 65] J. A. Robinson, "A Machine-Oriented Logic Based on the Resolution Principle," *Journal of the ACM* 12(1):23-41, January 1965.

[Robinson 71] J. A. Robinson, "Computational Logic: The Unification Computation," in B. Meltzer and D. Michie (Eds.), *Machine Intelligence, Vol. 6*, Edinburgh Univ. Press, Edinburgh, Scotland, 1971, pp. 63-72.

[Tarski 51] A. Tarski, *A Decision Method for Elementary Algebra and Geometry*, University of California Press, Berkeley, CA, 1951.

[Thiel 84] J.-J. Thiel, "A New Completeness Test," *Proc. of an NSF Workshop on the Rewrite Rule Laboratory, Sept. 6-9, 1983*, General Electric Corporate Research and Development Report No. 84GEN008, Schenectady, NY, April 1984, pp. 299-300.

[Yelick 84] K. A. Yelick, "A Generalized Approach to Equational Unification," Master's Thesis, MIT Lab. for Computer Science, 1984. (To appear.)

[Zachary 83] J. L. Zachary, "A Syntax-Directed Tool for Constructing Specifications," Master's Thesis, MIT Lab. for Computer Science, March 1983.